Magento U

Student Guide

Fundamentals of Magento 2 Development: Unit Six

magento.com/training/overview

Version 2.1

Magento® U

training@magento.com

magento.com/training/overview

magento.com/training/catalog/certification

Contents – Unit Six

About This Guide

This guide uses the following symbols in the notes that follow the slides.

Symbol	Indicates…
	A note, tip, or other information brought to your attention.
	Important information that you need to know.
	A cross-reference to another document or website.
	Best practice recommended by Magento

1. Unit Six Home Page

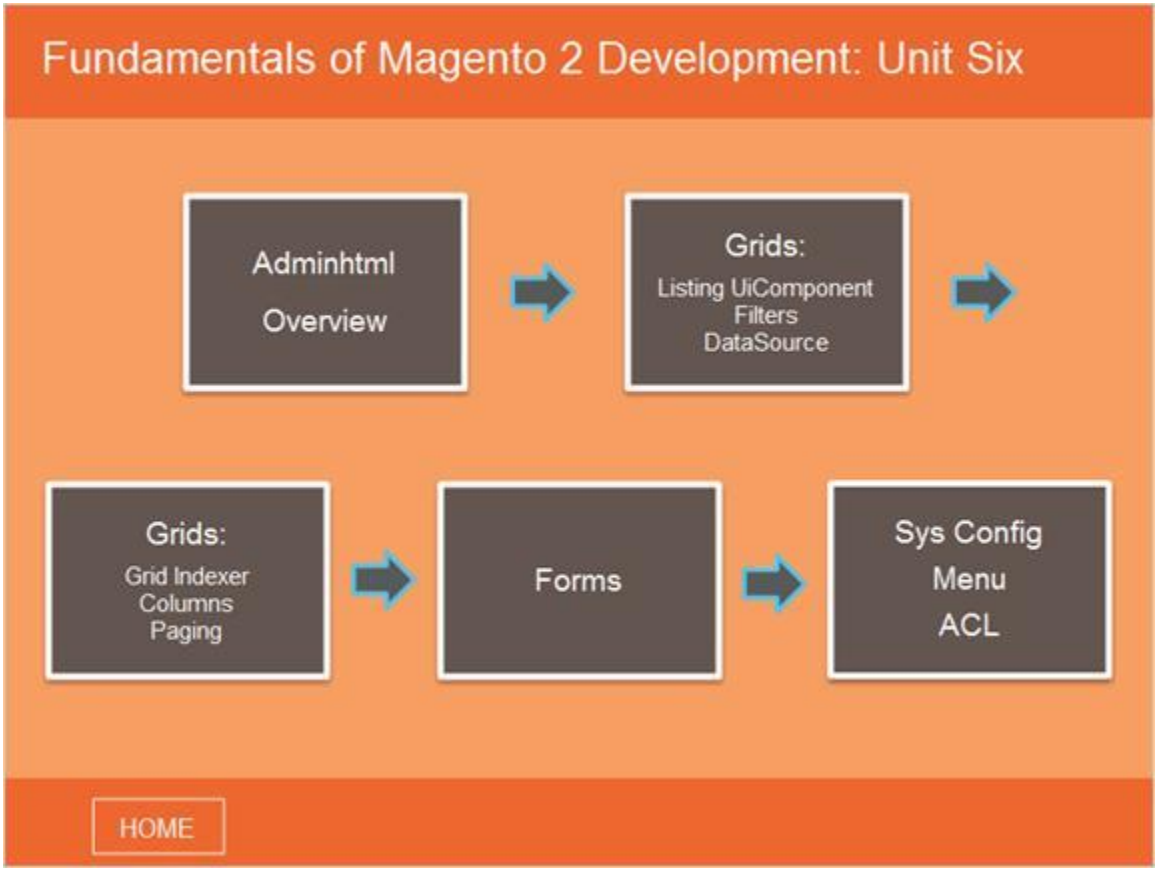
1.1 Fundamentals of Magento 2 Development - Unit Six

Notes:

Fundamentals of Magento 2 Development v. 2.1

Software version: Magento 2 v.2.1.0

1.2 Unit Six Home Page



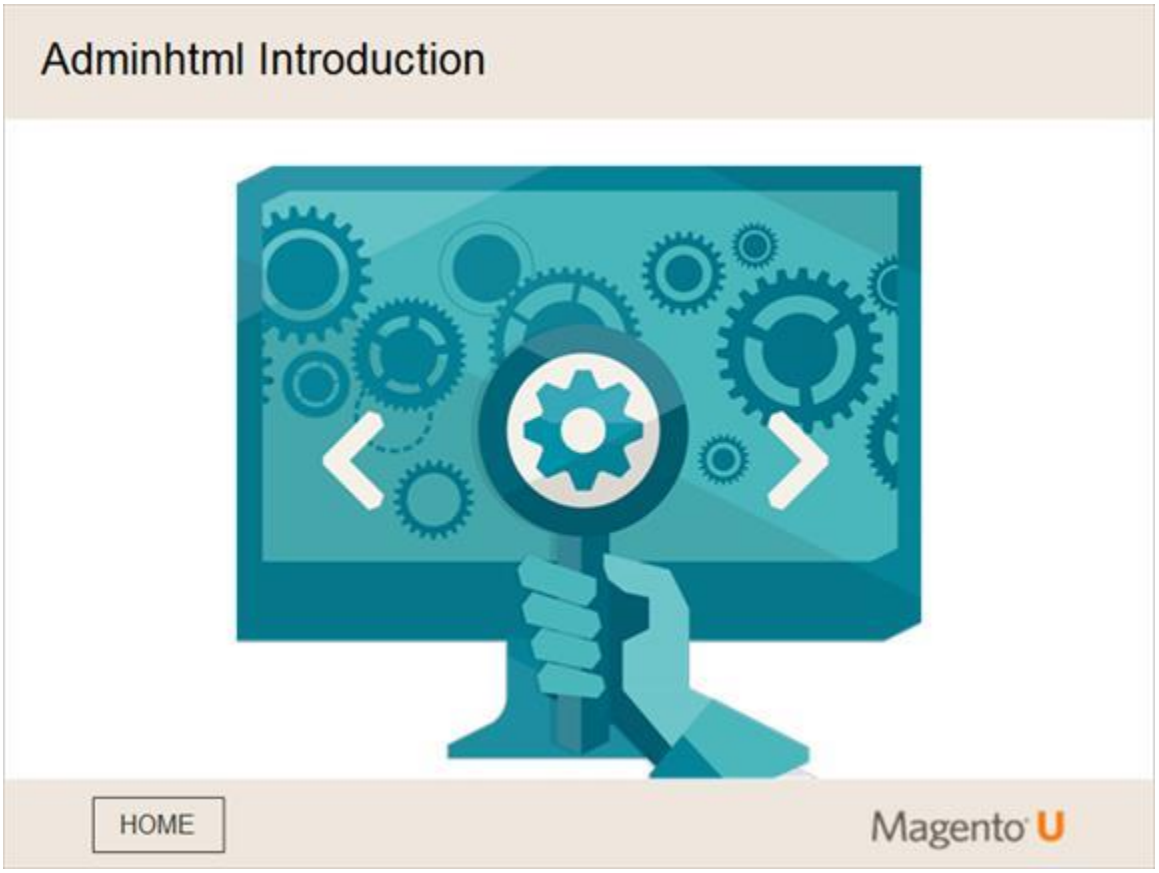
Notes:

Unit Six of the Magento 2 Fundamentals course contains five modules.

The suggested flow of the course is indicated by the arrows. However, you are free to access any of the modules, at any time, by simply clicking the Home button on the bottom of each slide.

2. Adminhtml Intro

2.1 Adminhtml Introduction



Notes:

This course unit provides an overview of the Adminhtml function in Magento 2.

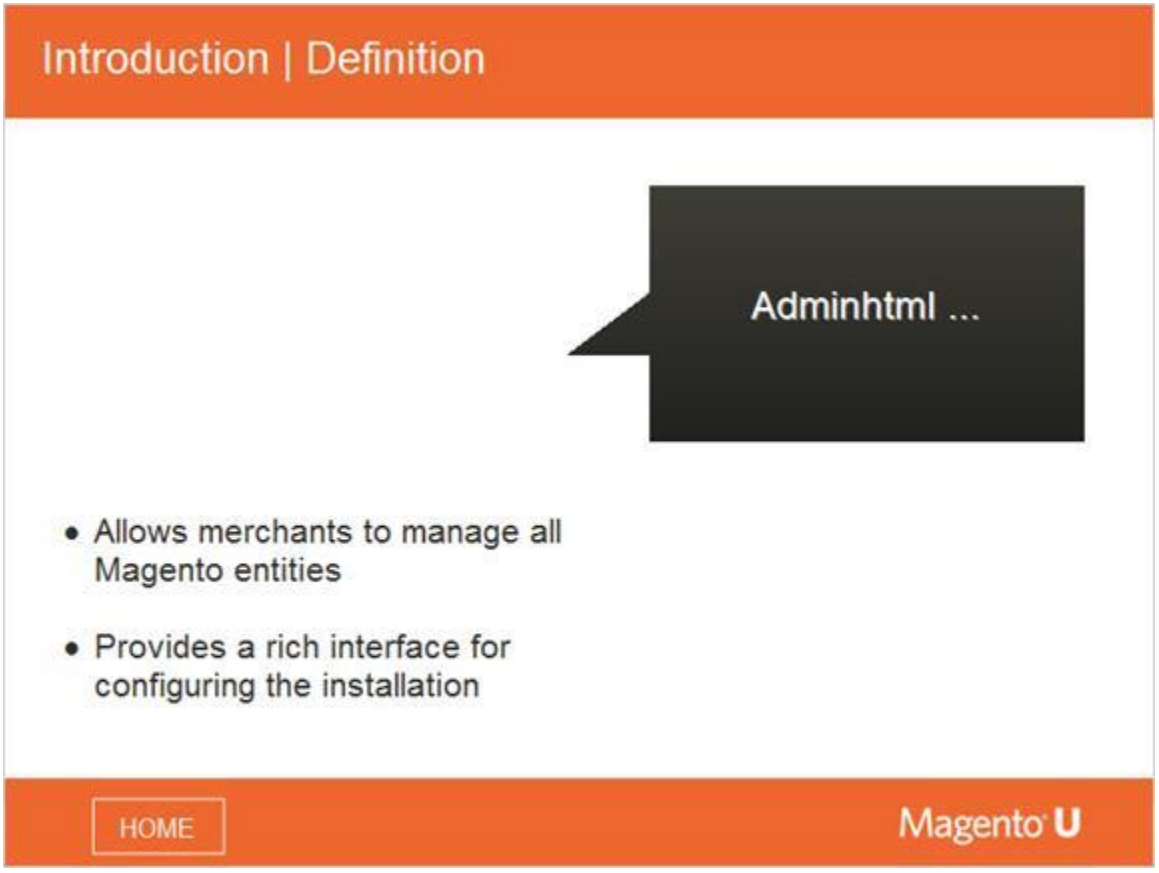
2.2 Module Topics | Adminhtml

Notes:

In this module, we will introduce you to Adminhtml in Magento 2.0, focusing on:

- Admin and Admin controller
- Grids
- Forms
- System configuration

2.3 Introduction | Definition

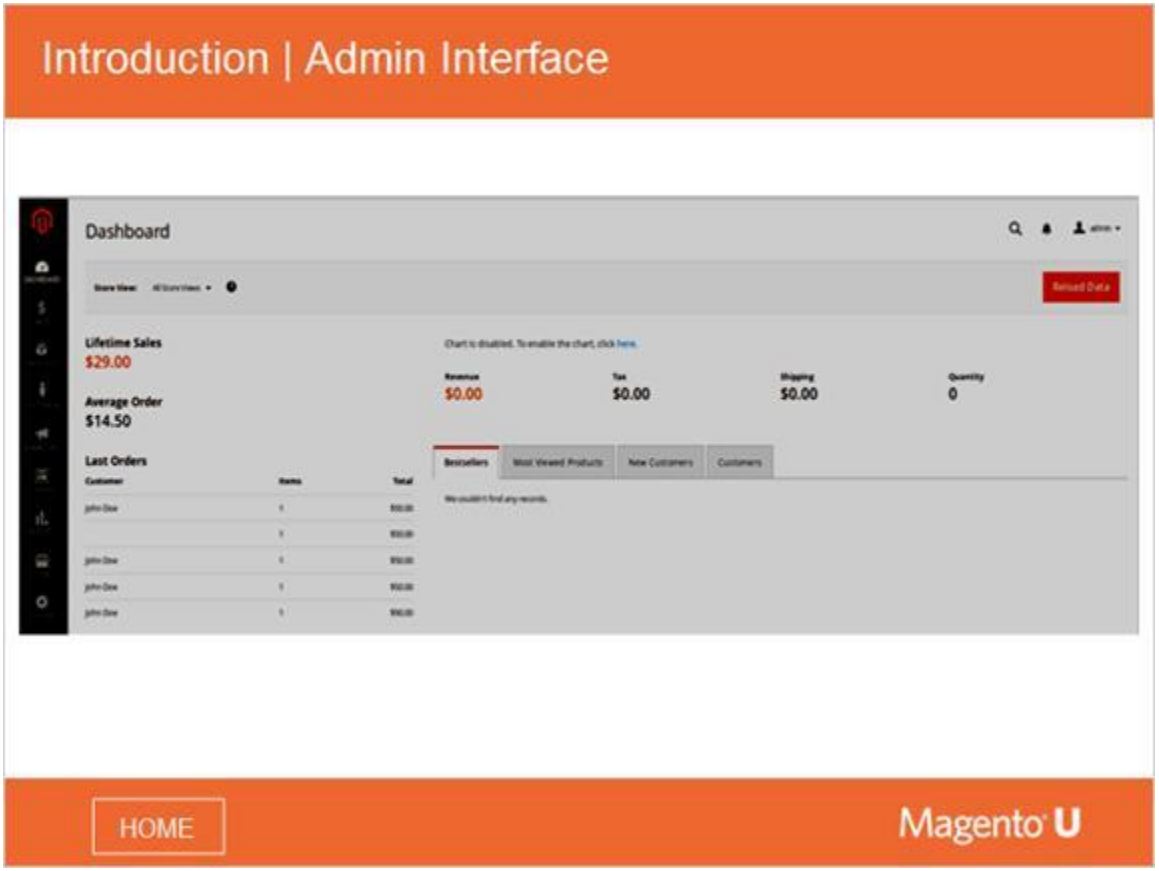


Notes:

The Adminhtml function of Magento 2 allows merchants and their staff to manage all Magento entities, such as products, categories, price rules, orders, and so on.

It also provides a rich interface for configuring your Magento installation.

2.4 Introduction | Admin Interface

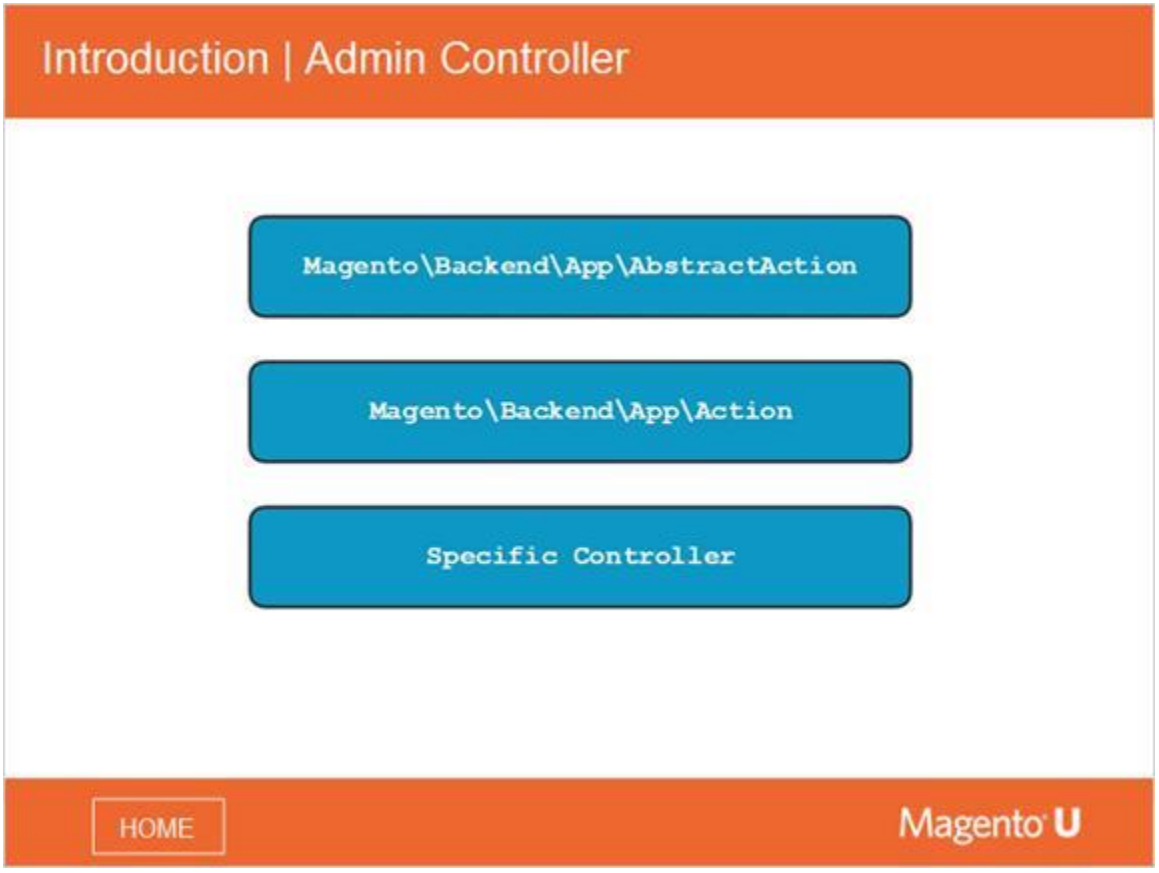


Notes:

Here is the Admin function in Magento 2.

In this unit we will discuss the Admin architecture, specifically grids, forms, system configuration, and ACL.

2.5 Introduction | Admin Controller

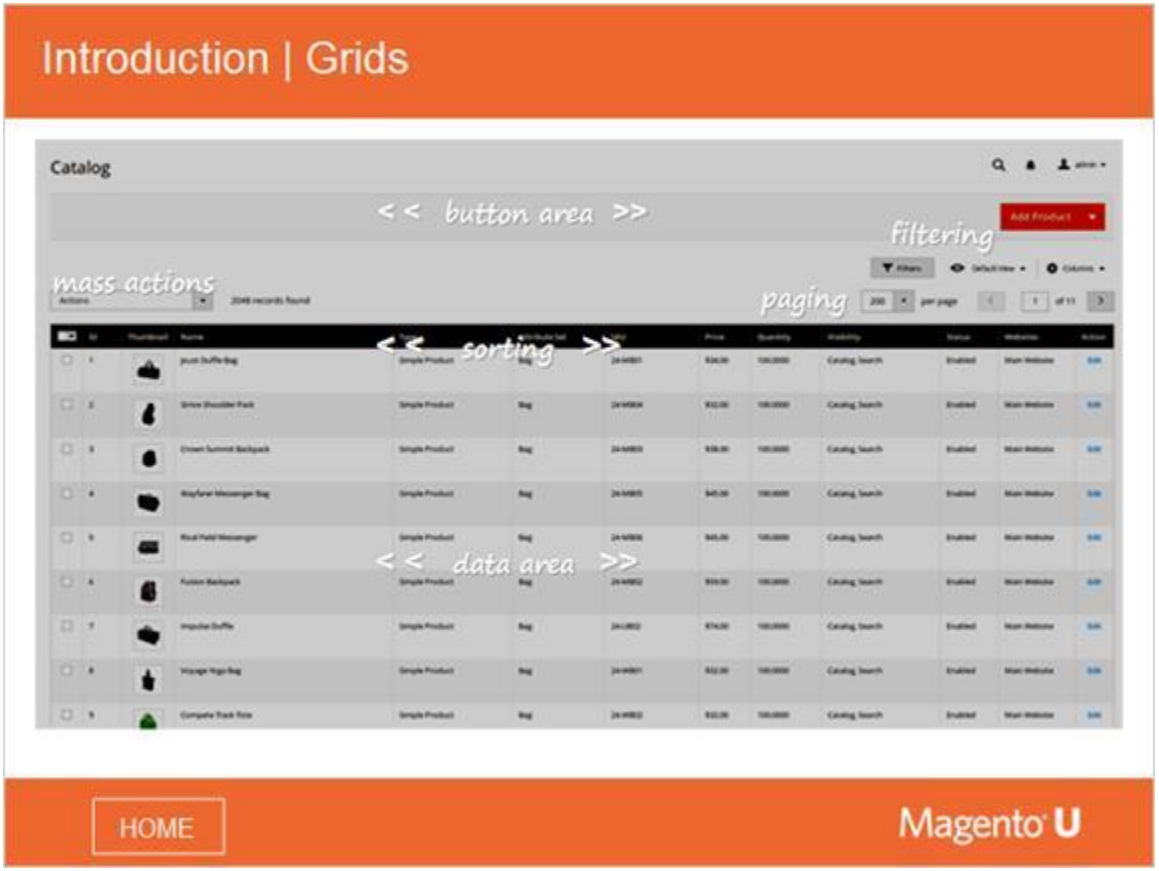


Notes:

All admin controllers extend the `Magento\Backend\App\Action` class, which in turn extends `Magento\Backend\App\AbstractAction` (and the latter extends the standard `Magento\Framework\App\Action\Action`).

Each admin controller has to extend this class -- `Magento\Backend\App\Action` -- in order to provide backend-specific functionality, such as secret key processing and ACL mechanism implementation.

2.6 Introduction | Grids

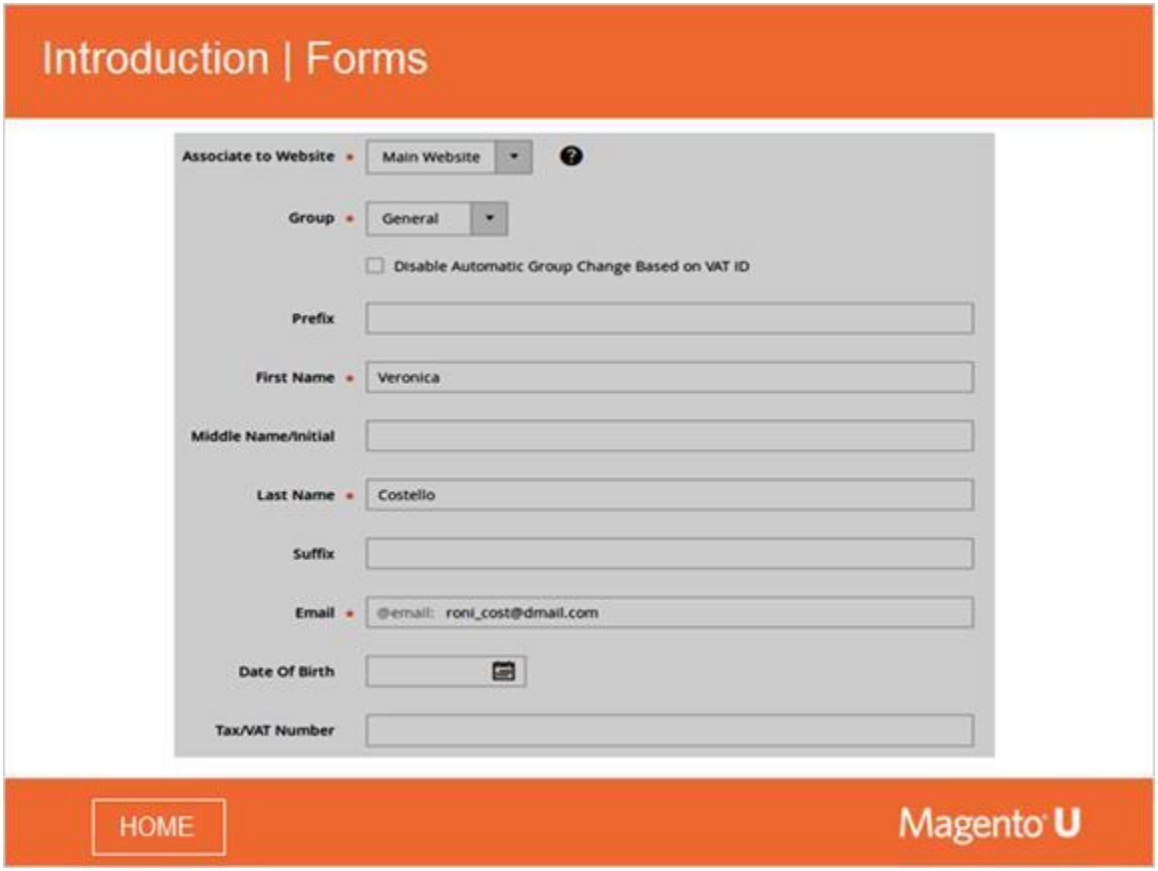


Notes:

Grid is a UiComponent that represents a list of records for a specific entity.

Grids in Magento 2 have significantly changed when compared to Magento 1. They have many new functions, which will be covered in more detail in the Grids module that follows.

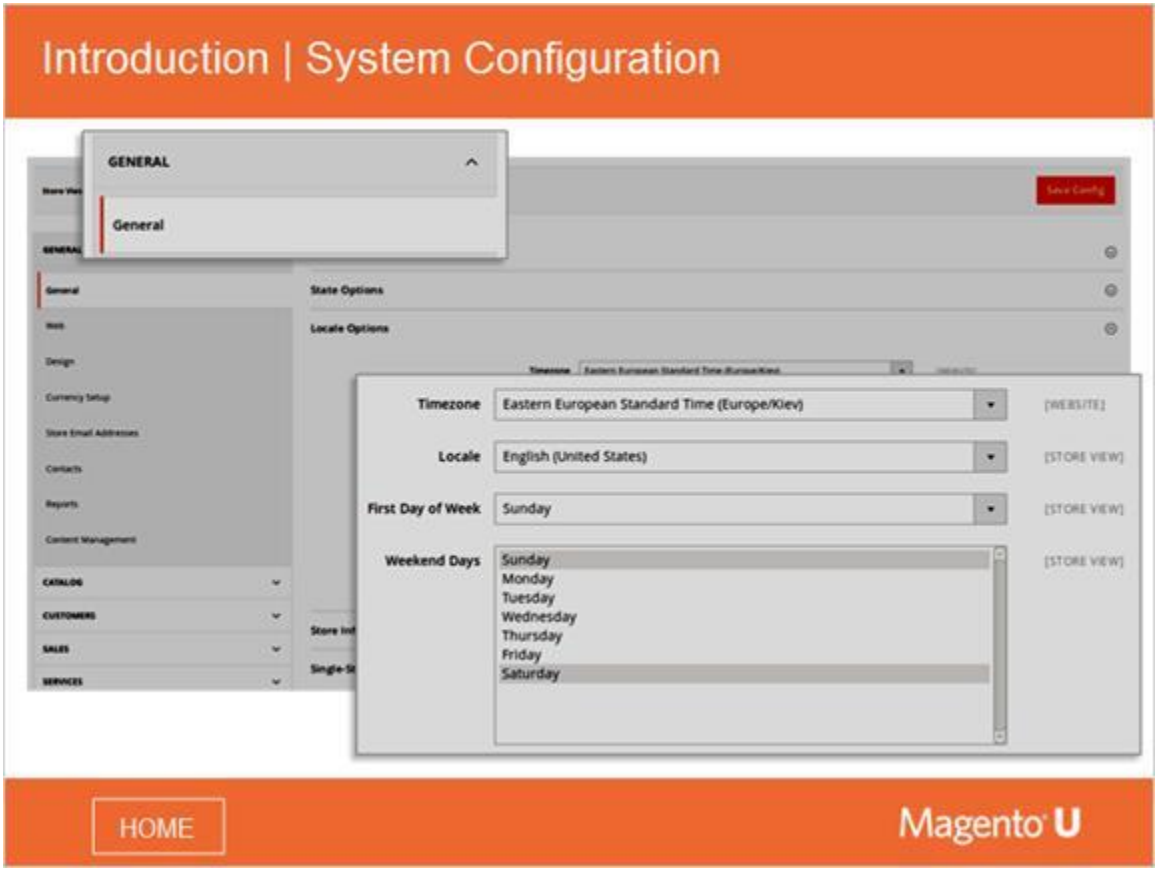
2.7 Introduction | Forms



Notes:

Forms are used to allow editing of a certain entity. They are represented by UiComponents as well as grids.

2.8 Introduction | System Configuration



Notes:

As in Magento 1, there is a highly customizable page in Magento 2 where a merchant can add any configuration options required for the installation.

All related values will be stored in a database, and can have different scopes: global, website, store. As this page is used by the merchant, not a developer, all the available options should be clearly presented.

Options that a developer would modify should only be in the XML configuration.

3. Grids Part 1

3.1 Grids: Part 1

Notes:

This course unit discusses the Adminhtml function in Magento 2, starting with Grids.

3.2 Module Topics | Grids

Notes:

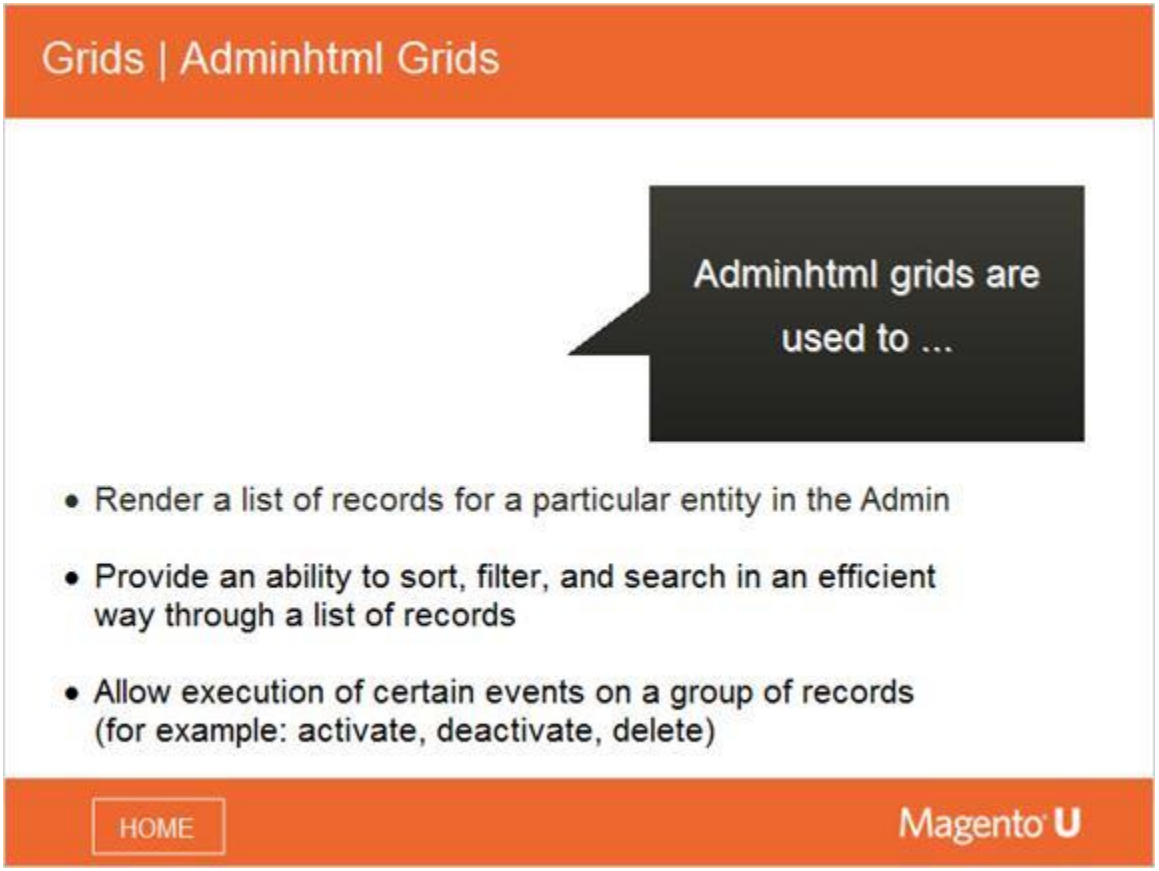
In this module, we will provide an overview of Adminhtml grids, focusing on the topics of listing UiComponents, filters, DataSource, the grid indexer, columns, and paging.

3.3 Grids | Overview

Notes:

We'll begin with a short review of grids in Magento 2.

3.4 Grids | Adminhtml Grids



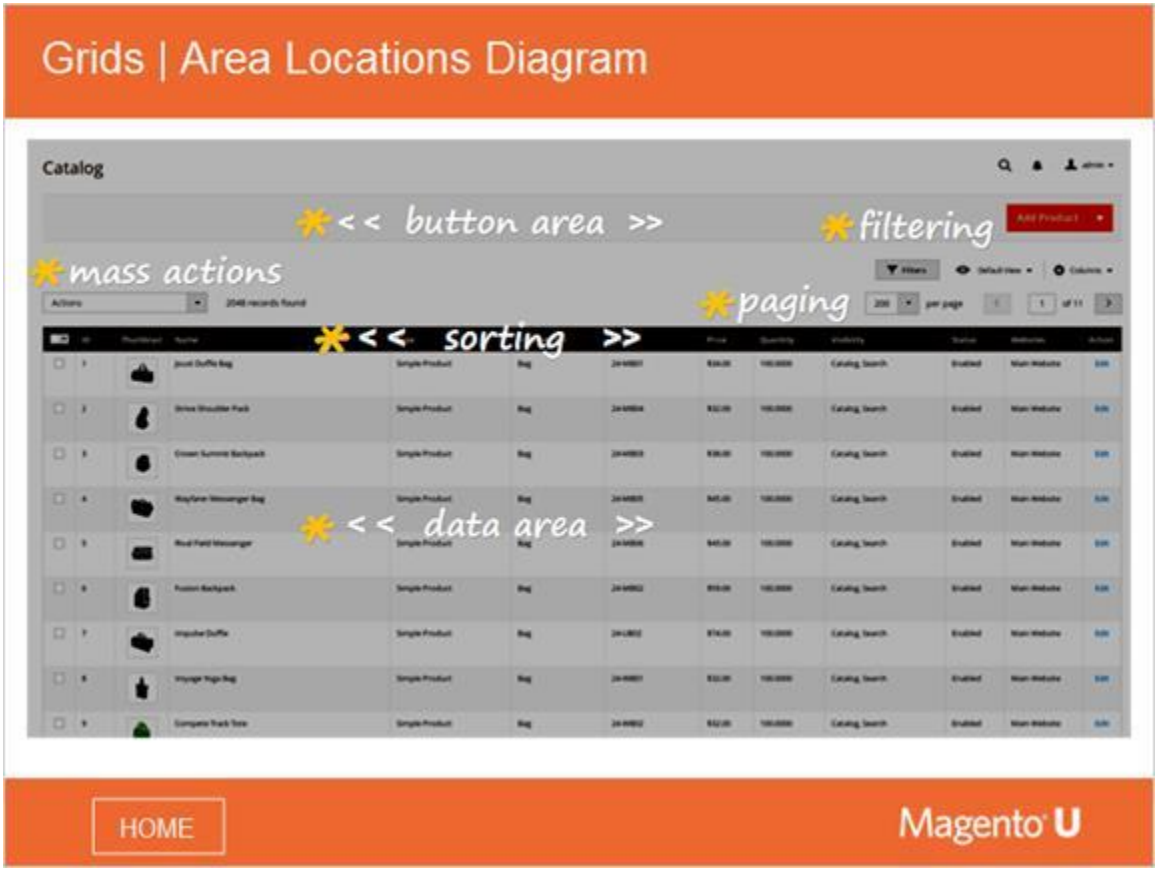
Notes:

Adminhtml grids fulfill a number of important functions, such as:

- Render a list of records for a particular entity in admin.
- Provide an ability to sort, filter, and search in an efficient way through a list of records.
- Allow execution of certain events on a group of records (for example: activate, deactivate, delete).

We will look at each of these topics within this section.

3.5 Grids Area Locations Diagram



Notes:

On this slide, you can see what a grid looks like in Magento 2 (which is quite different from Magento 1).

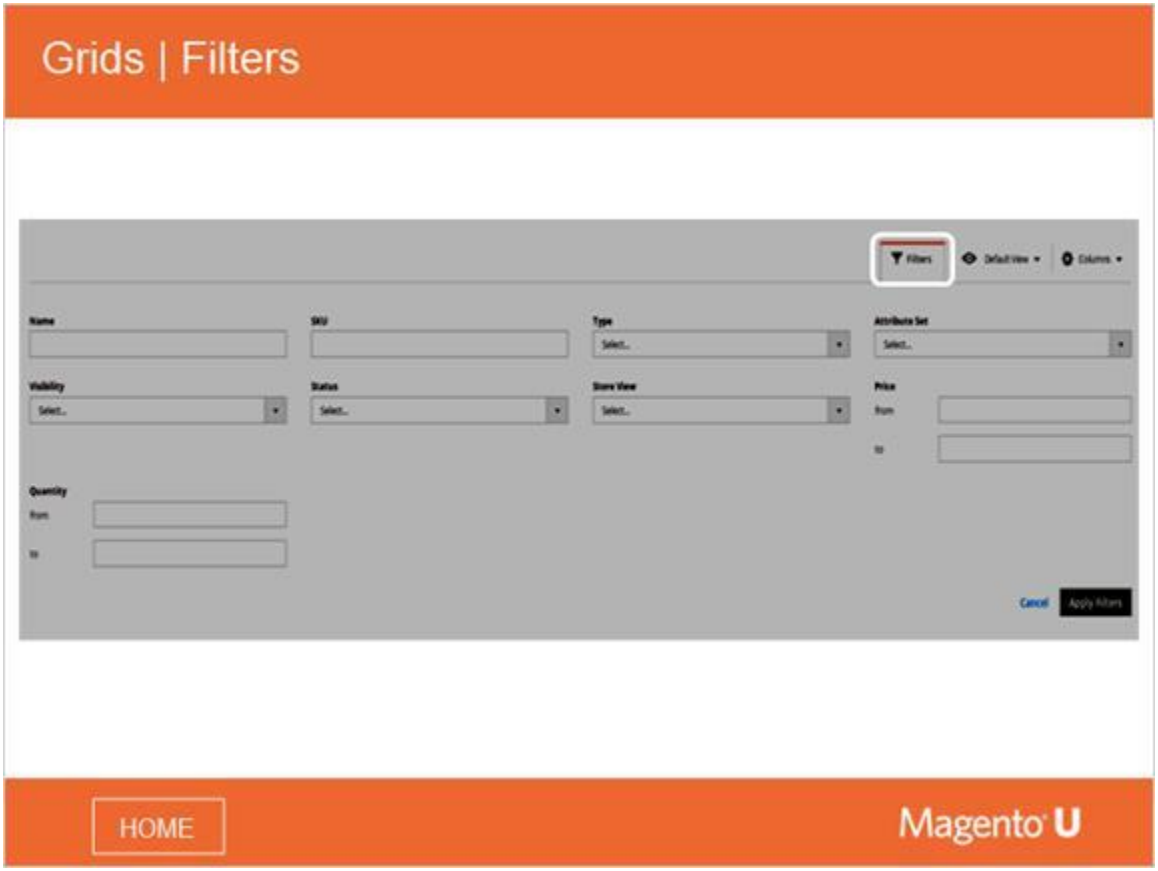
The first thing to notice is the filtering function. In Magento 1, there are filters at the top of each column; here in Magento 2, filters are now separated from the data.

You can also see other element areas: buttons, mass actions, paging, sorting, and data.

In this section we will review all these grid components.

In general, grid implementation in Magento 2 has been significantly changed from Magento 1. Now grid is a UiComponent, rendered by JavaScript. A developer only has to configure it properly.

3.6 Grids | Filters



Notes:

This slide shows the expanded filters feature. This interface opens when the “Filter” button is clicked.

Each filter is represented by its own input, which provides greater flexibility and the creation of more sophisticated filters.

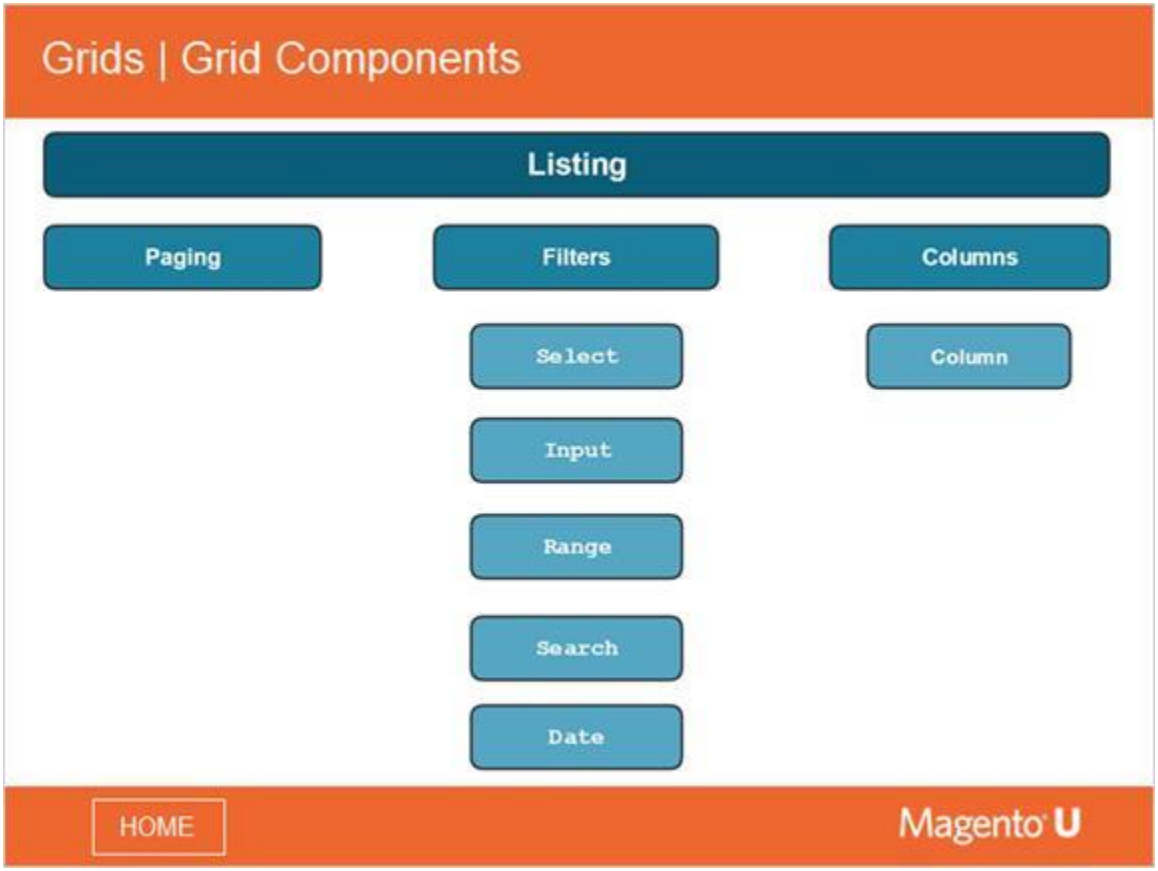
Filters are covered in more detail later in this module.

3.7 Grids | layout.xml

Notes:

This code demonstrates how a grid is included into a page. As mentioned earlier, a grid is a UiComponent in Magento 2, so it should be enabled using the `<uiComponent>` layout directive.

3.8 Grids | Listing UiComponents



Notes:

This diagram presents a list of the various grid components:

Listing ... a central UiComponent that represents the grid itself.

Columns ... a component that includes a list of “Column” components. Each “Column” component represents a specific grid column and its data. As columns are now rendered in JavaScript, their functionality has been greatly expanded over Magento 1.

Filters ... a composite component that includes a set of “Filter” components. The diagram displays which native filter components are available in a native Magento installation.

Paging ... a component responsible for rendering paging.

Massactions ... a component that renders a massactions dropdown menu with a list of possible actions appropriate to a subset of a grid’s records.

3.9 Grids | Listing UiComponent

Notes:

We'll now look at the Listing UiComponent in more detail.

3.10 ListingUiComponent | Overview

Notes:

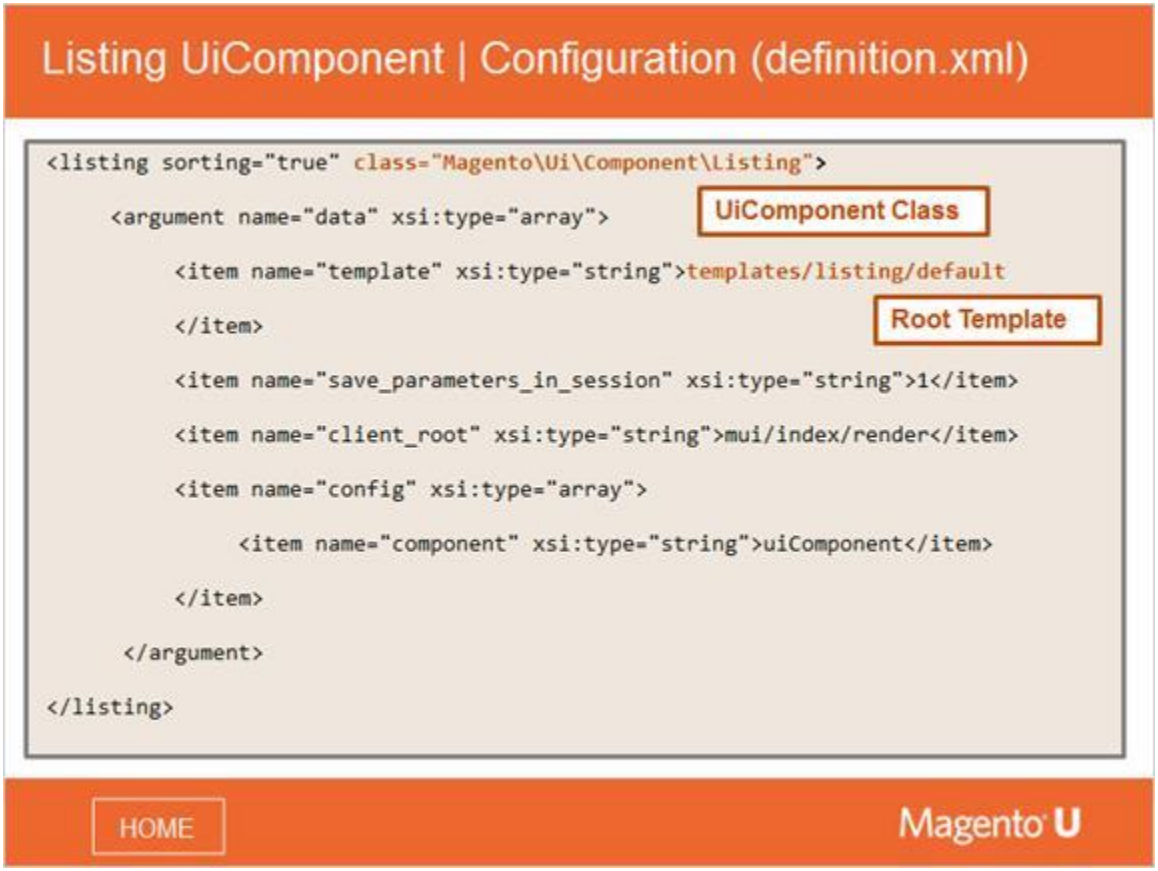
The Listing UiComponent is:

- An analog of the Grid widget in Magento 1.
- Has a group of children to facilitate its features.
- Still works with collections (like Magento 1) through the `DataProvider`.

> **Note:** This may change in future releases.

We will look at each of these topics within this section.

3.11 Listing | Configuration (definitions.xml)

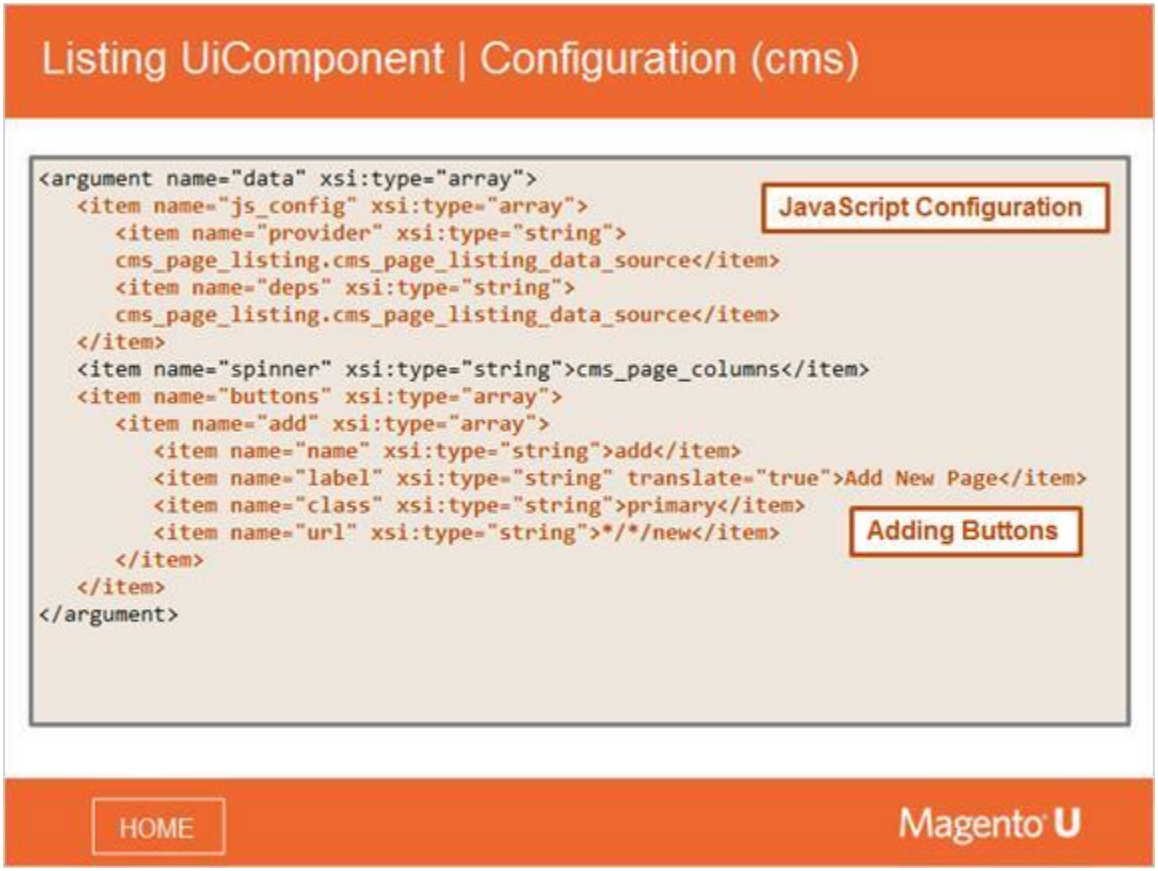


Notes:

A code demonstration of the Listing configuration from a `definition.xml` file.

Note the UiComponent class -- `Magento\Ui\Component\Listing`, and the root template -- `templates/listing/default.xhtml`. This template is generic and is used like a container; all real rendering occurs by using HTML templates and JavaScript.

3.12 Listing UiComponent | Configuration (cms)



Notes:

In this code, you can see an example from the Cms module that demonstrates arguments a listing can use.

The first highlighted code shows the `js_config` argument -- it defines the configuration for JavaScript.

The second highlight shows a `buttons` argument, which specifies a list of buttons that can be added to the listing.

3.13 Grids | Filters

Notes:

Now, a further discussion of filters.

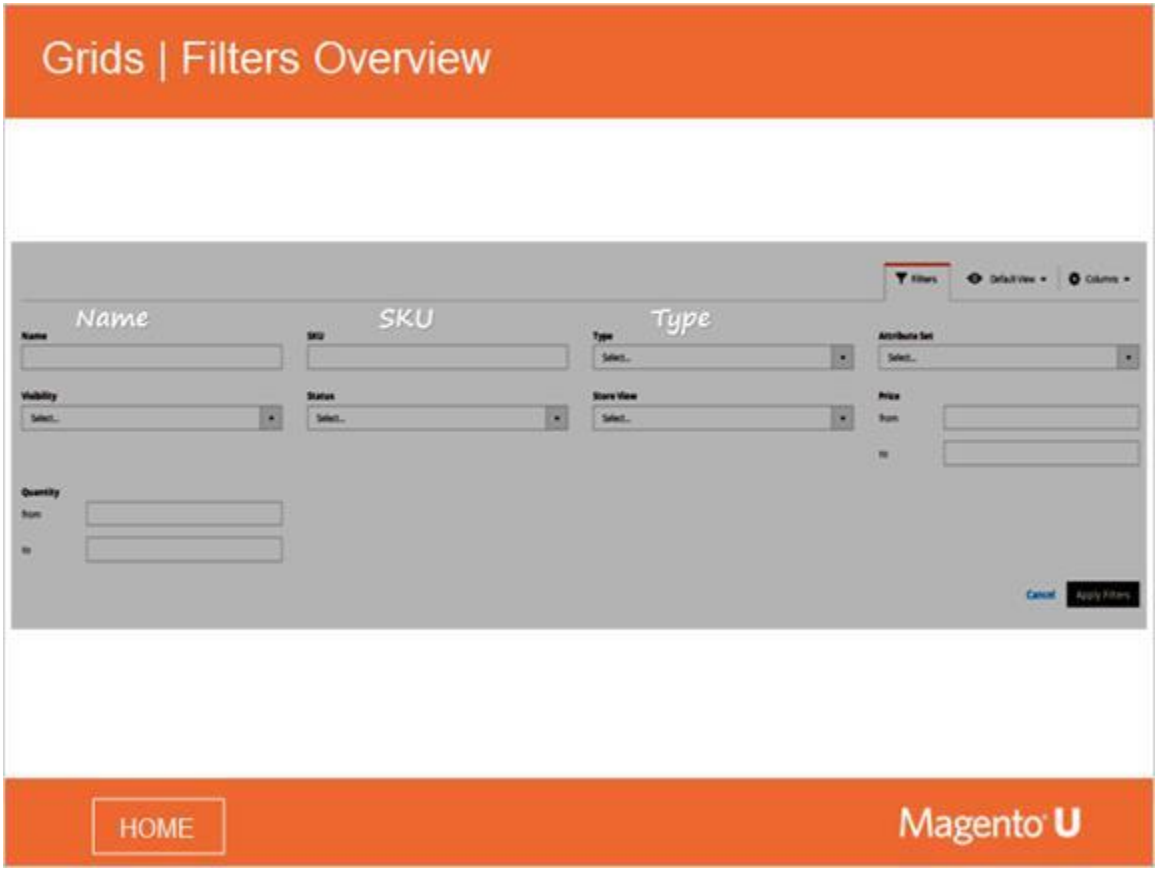
3.14 Filters | Overview

Notes:

Filters are used to simplify searching through a list of records.

They can also set limits to display only those records that meet a set of criteria set by the Admin.

3.15 Grids | Filters Overview



Notes:

This slide again shows the expanded filters interface, first presented in the Grids overview.

Having the filters separated means you can now have more filters than visible fields, and the inputs for the filters can be complex combinations. For example, on the image you can see Name, SKU, and Type.

As filters are UiComponents, a list of all available filters can be found in `definition.xml`.

3.16 Filters | UiComponent

Notes:

This code snippet shows the “filters” UiComponent configuration in `default.xml`.

Note the `filters` UiComponent and the set filter component; `filters`, act as a container for all specific filters.

Also on the slide you can see the JavaScript module, which renders this UiComponent.

3.17 Filters | Product Listing Example

Notes:

This slide presents an example of a Filters PHP class implementation for a products grid (`Magento\Catalog\Ui\Component\Listing\Filters`).

The code demonstrates how a list of filters is generated based on the product attributes.

3.18 Filters | Product Listing Example

Notes:

How filters are applied depends upon the data provider implementation (which will be covered later).

The native data provider (`\Magento\Framework\View\Element\UiComponent\DataProvider\DataProvider`) uses a `FilterPool` object (`\Magento\Framework\View\Element\UiComponent\DataProvider\FilterPool`), which contains an `applyFilters` method (shown on the slide) that applies filters.

Configuration of the filter appliers seen on the slide can be defined in the `di.xml` for a specific grid.

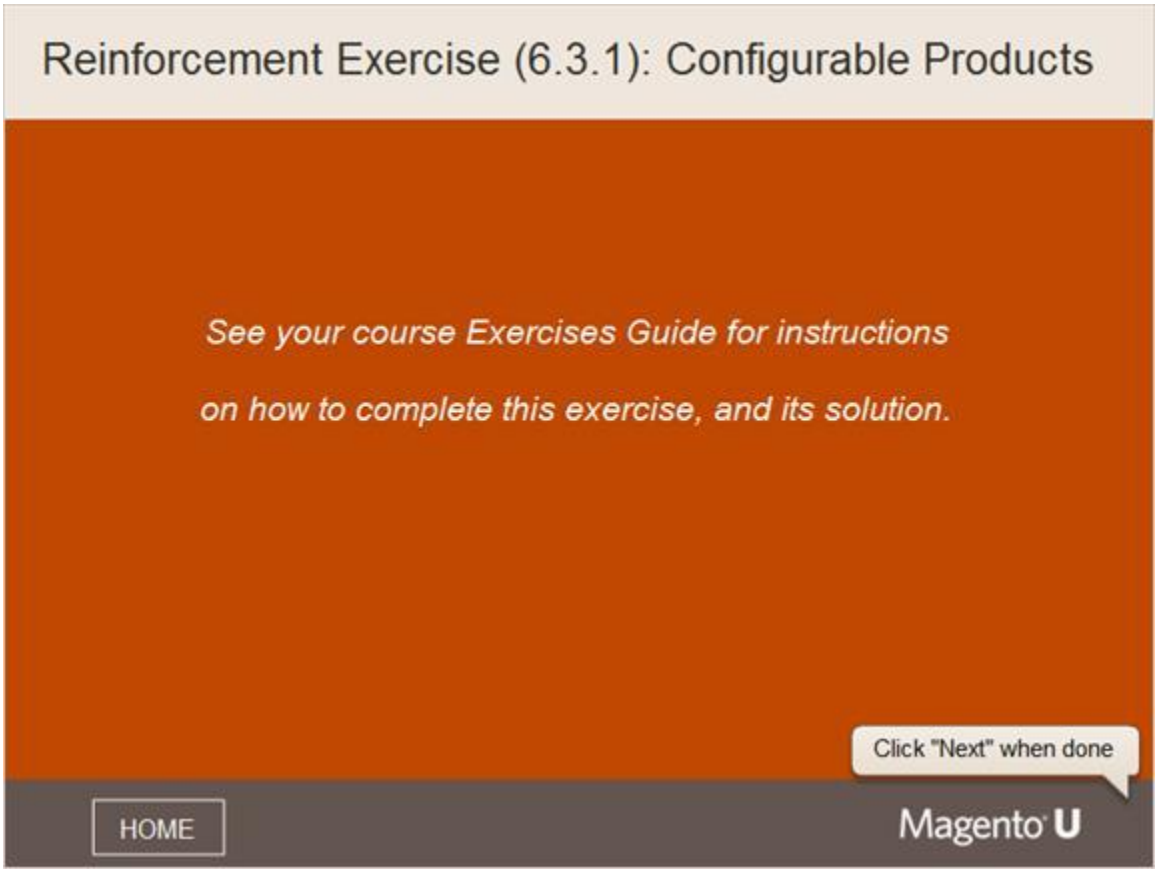
3.19 Filters | Filter Pool

Notes:

As you can see from this second code snippet, the process goes through a list of registered appliers, which it uses to apply a filter.

In general, the list of appliers is injected using DI.

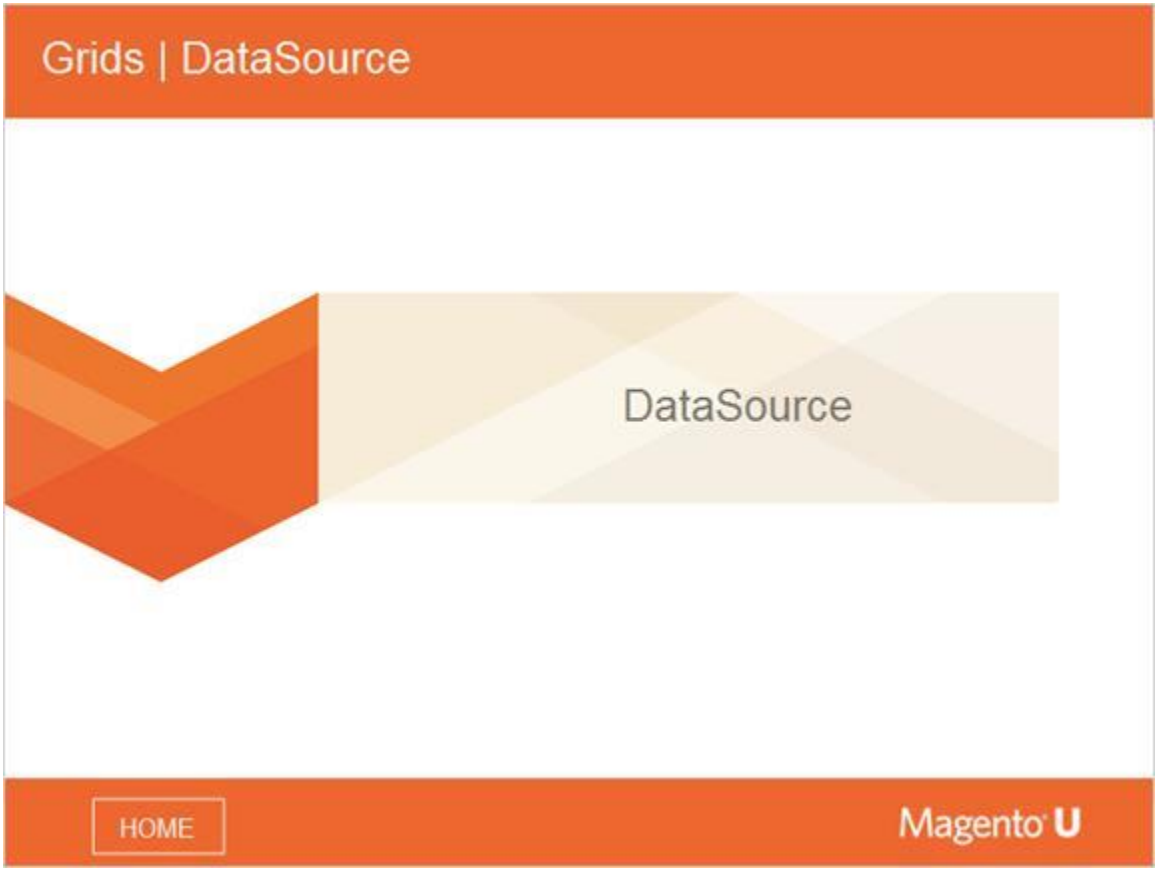
3.20 Reinforcement Exercise (6.3.1)



Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

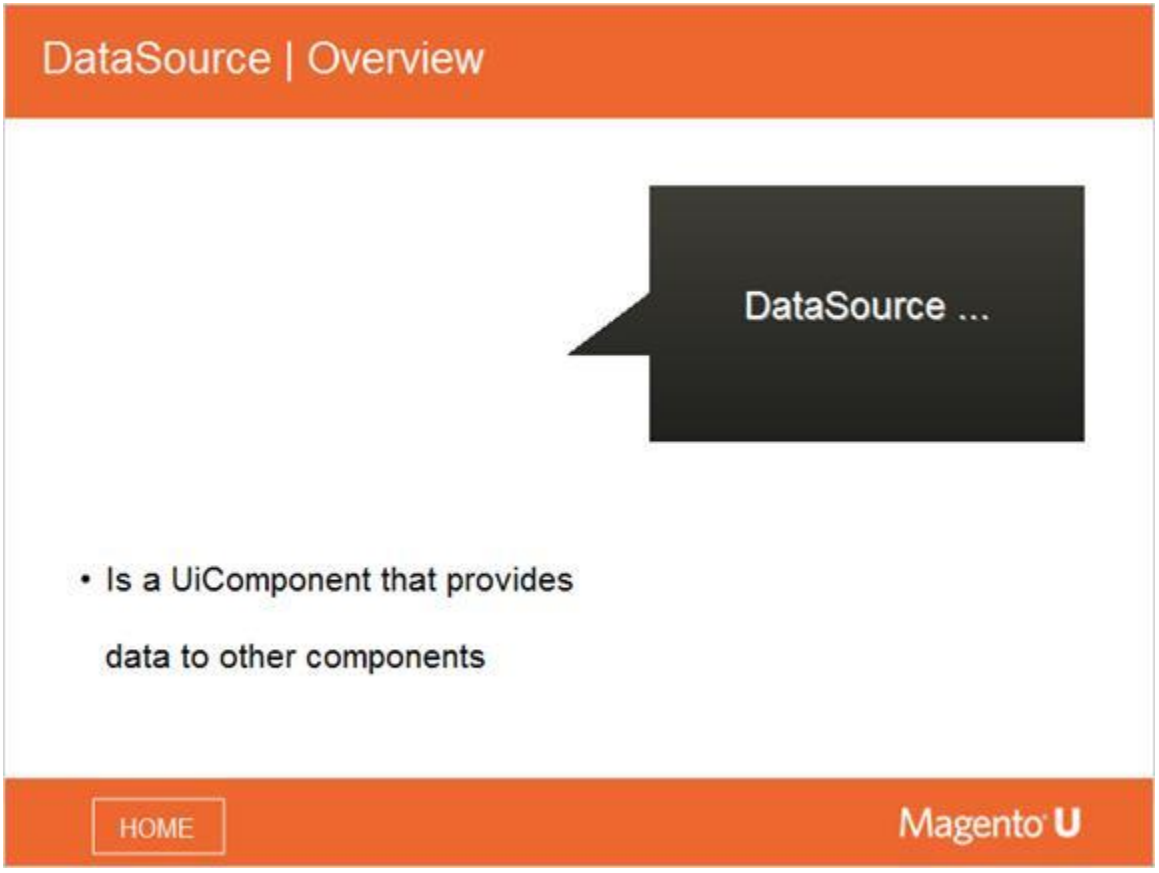
3.21 Grids | DataSource



Notes:

The next topic we'll cover is DataSource.

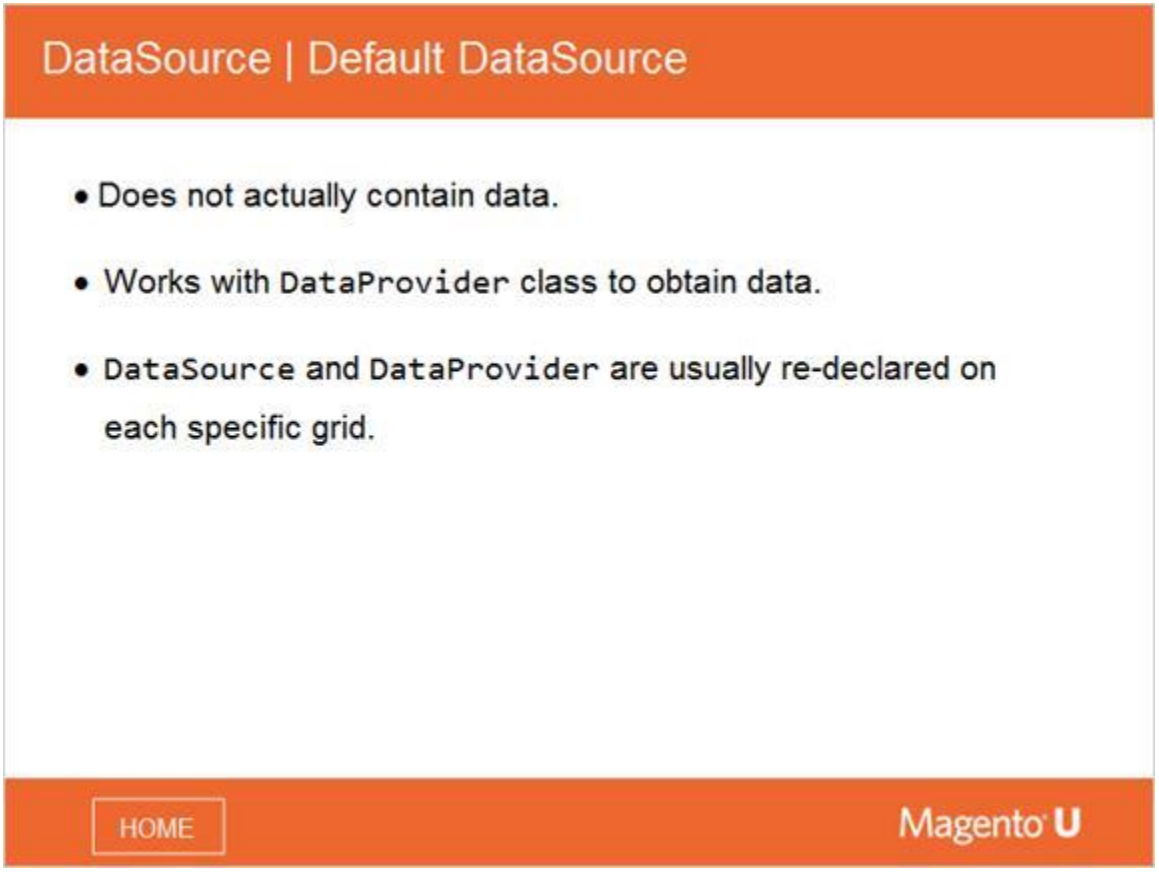
3.22 DataSource | Overview



Notes:

DataSource is a UiComponent that provides data to other components.

3.23 DataSource | Default DataSource



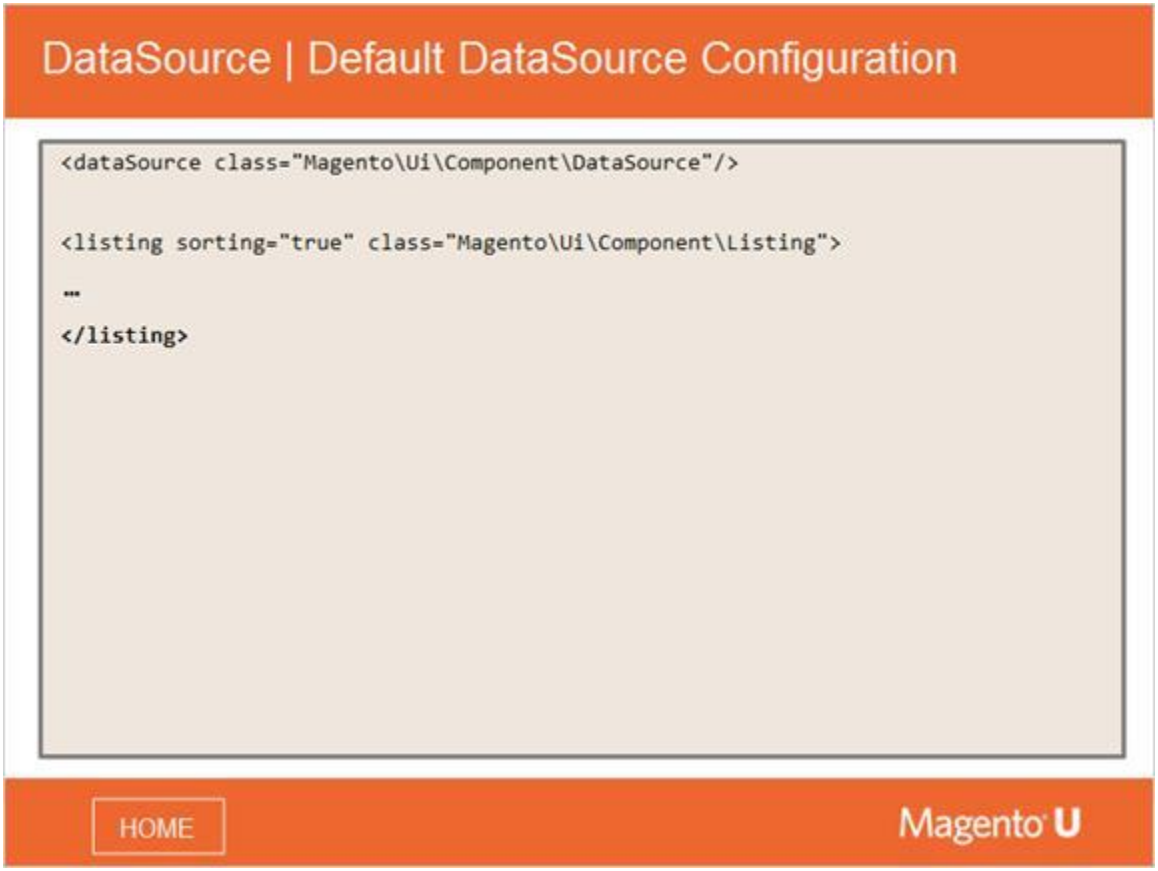
Notes:

As discussed earlier, `DataSource` requires `DataProvider`. `DataProvider` is actually the class which obtains data. It has to implement a specific interface to provide `JSON` data to DataSource.

You usually implement your own `DataProvider` for an instance of a UiComponent.

We will learn more about `DataProviders` on the slides that follow.

3.24 DataSource | Default DataSource Configuration



Notes:

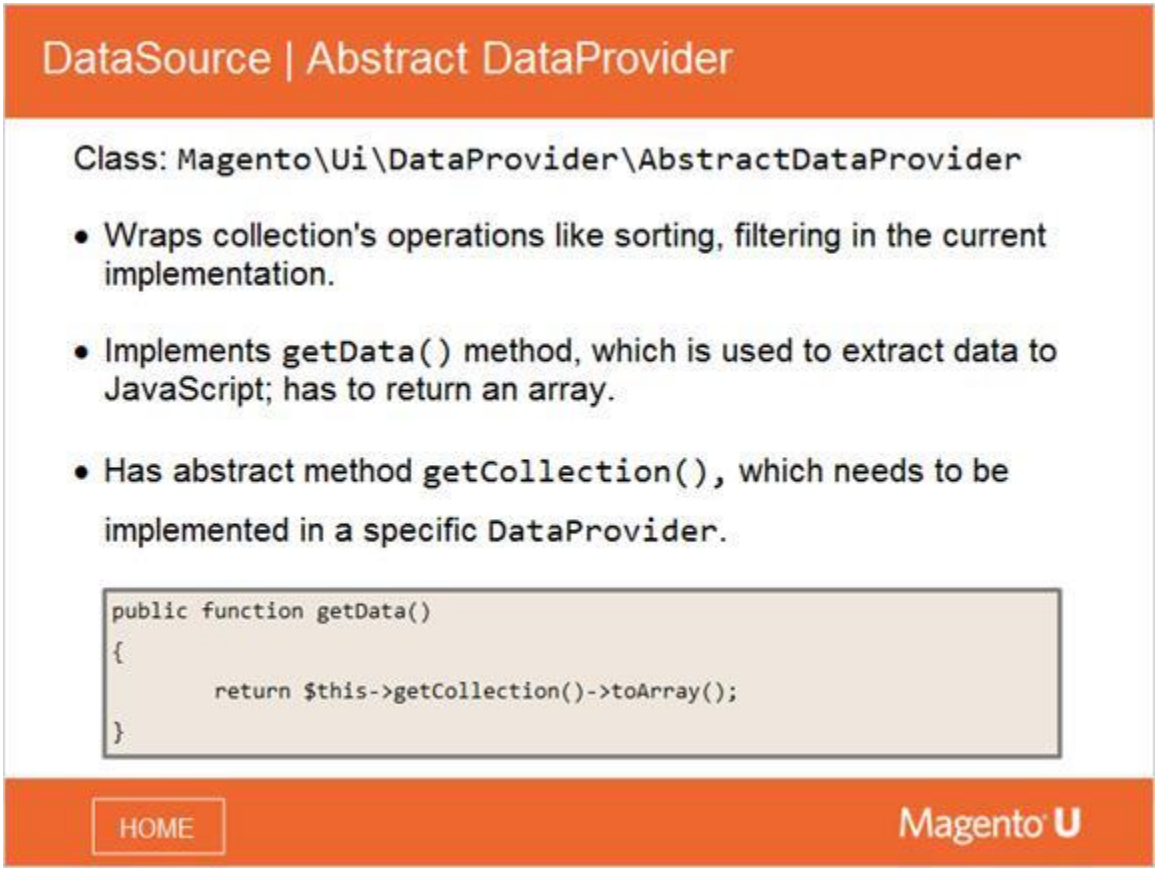
There are a few commonly used UiComponents available in Magento 2. Two of them -- `listing` and `form` -- are composite and contain a set of child components.

`DataSource` is also a UiCcomponent that provides data to all of them.

Of course, data needs to be taken from the database. But `DataSource` is a JavaScript module, so it uses `DataProvider`, which is a PHP class, to generate data for `DataSource`.

The code on this slide demonstrates configuration for the `DataSource` component in the `definition.xml` file.

3.25 DataSource | Default DataSource



Notes:

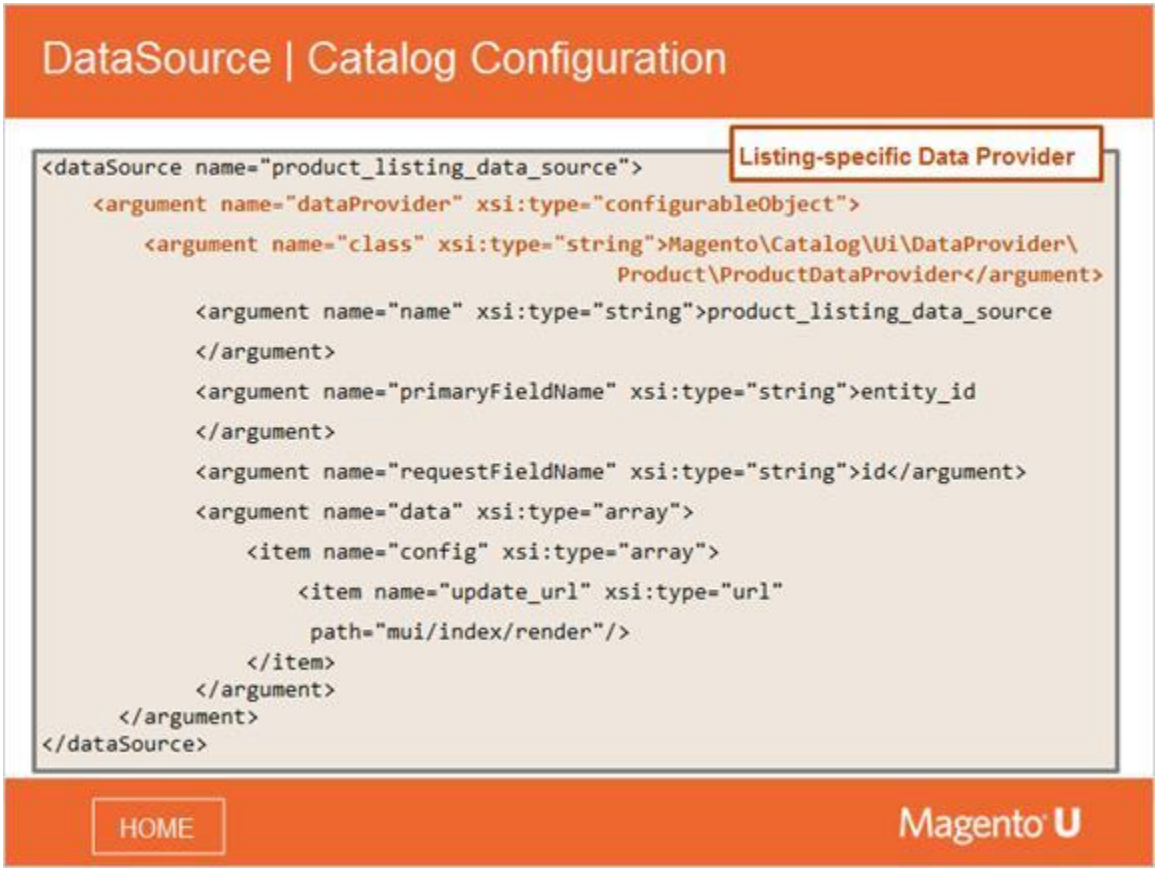
`AbstractDataProvider` is a class that every `DataProvider` has to extend; its key method `getData` has to be implemented.

On the slide, you can see an example of this type of implementation. The `getData` method returns data from a collection.

This is typical of grids -- their `DataProvider` classes usually only wrap collection classes, which extract data from the database.

Note: There is another abstract data provider available: `\Magento\Framework\View\Element\UiComponent\DataProvider\DataProvider`. Those classes are used in different places in the core code.

3.26 DataSource | Catalog Configuration

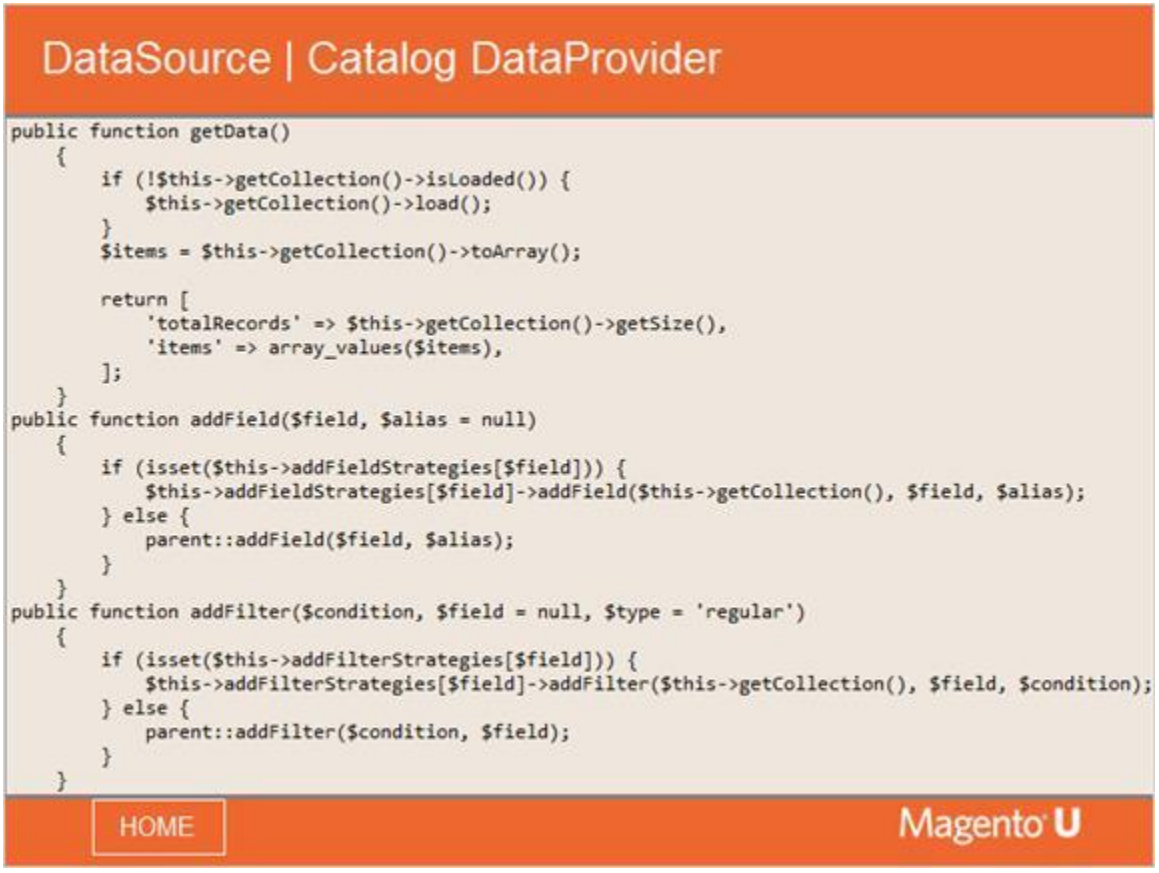


Notes:

Here is an example of the `DataSource` component configuration for product listing.

You can see that it defines the `DataProvider` class. Usually, the `DataProvider` class is defined for each implementation of a UiComponent.

3.27 DataSource | Catalog DataProvider



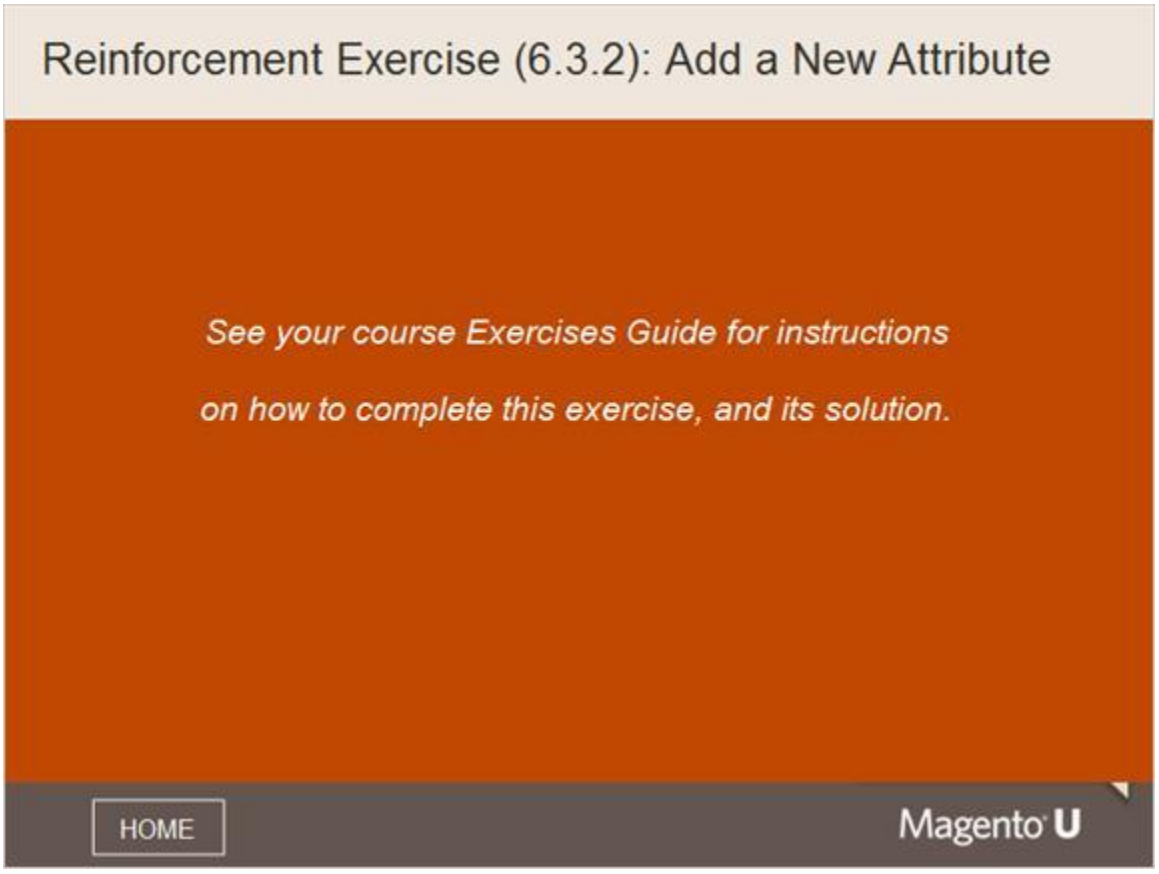
Notes:

This slide shows an implementation of `DataProvider` for the `product_listing` UiComponent. As you can see, `getData` loads a collection and creates an array with “totalRecords” and “items” elements.

You can also see a couple of other methods in the code; for example, `addFilter()`. It is just a wrapper around collection filtering.

The `DataSource` Catalog `DataProvider` implements three methods:
- `getData()`
- `addField()`
- `addFilter()`

3.28 Reinforcement Exercise (6.3.2)



Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

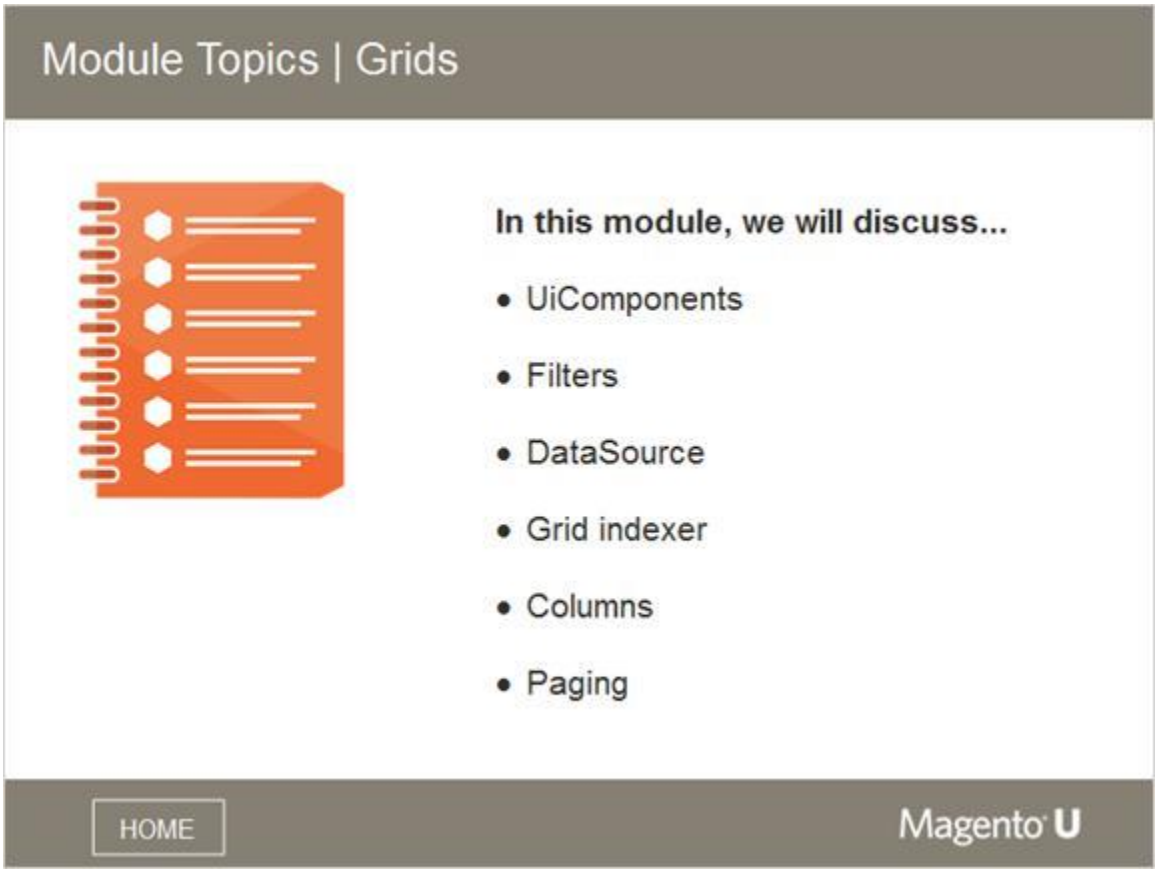
4. Grids Part 2

4.1 Grids: Part 2

Notes:

We now continue our discussion of grids.

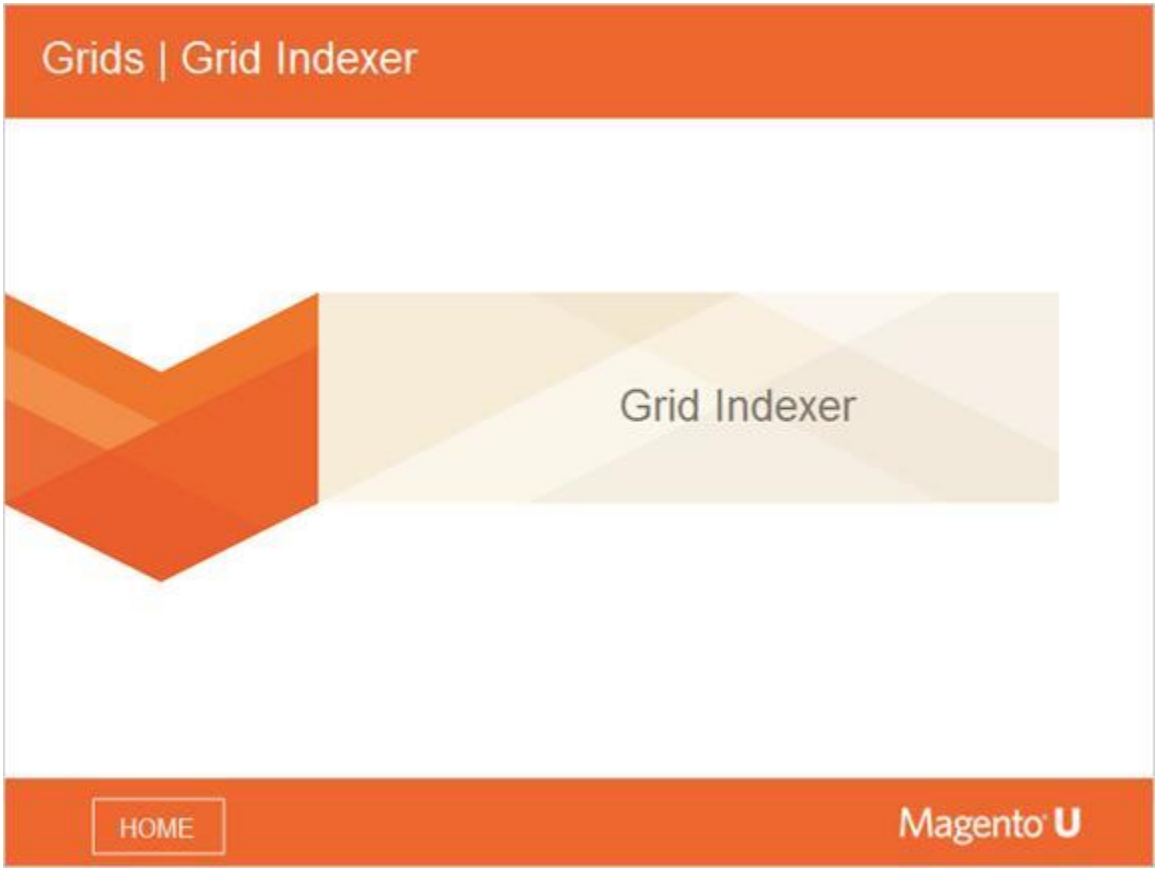
4.2 Module Topics | Grids



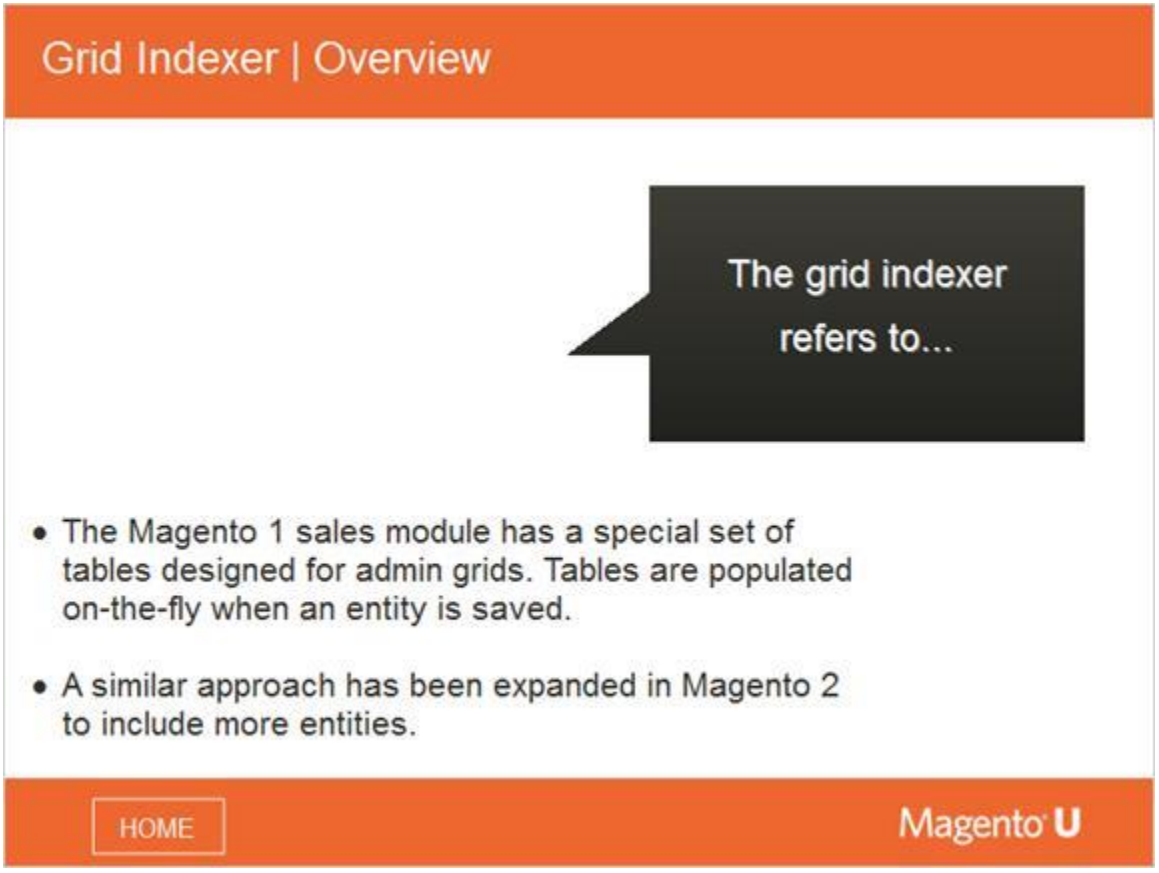
Notes:

In this module, we will focus on listing UiComponents, filters, DataSource, the grid indexer, columns, and paging.

4.3 Grids | Grid Indexer



4.4 Grid Indexer | Overview

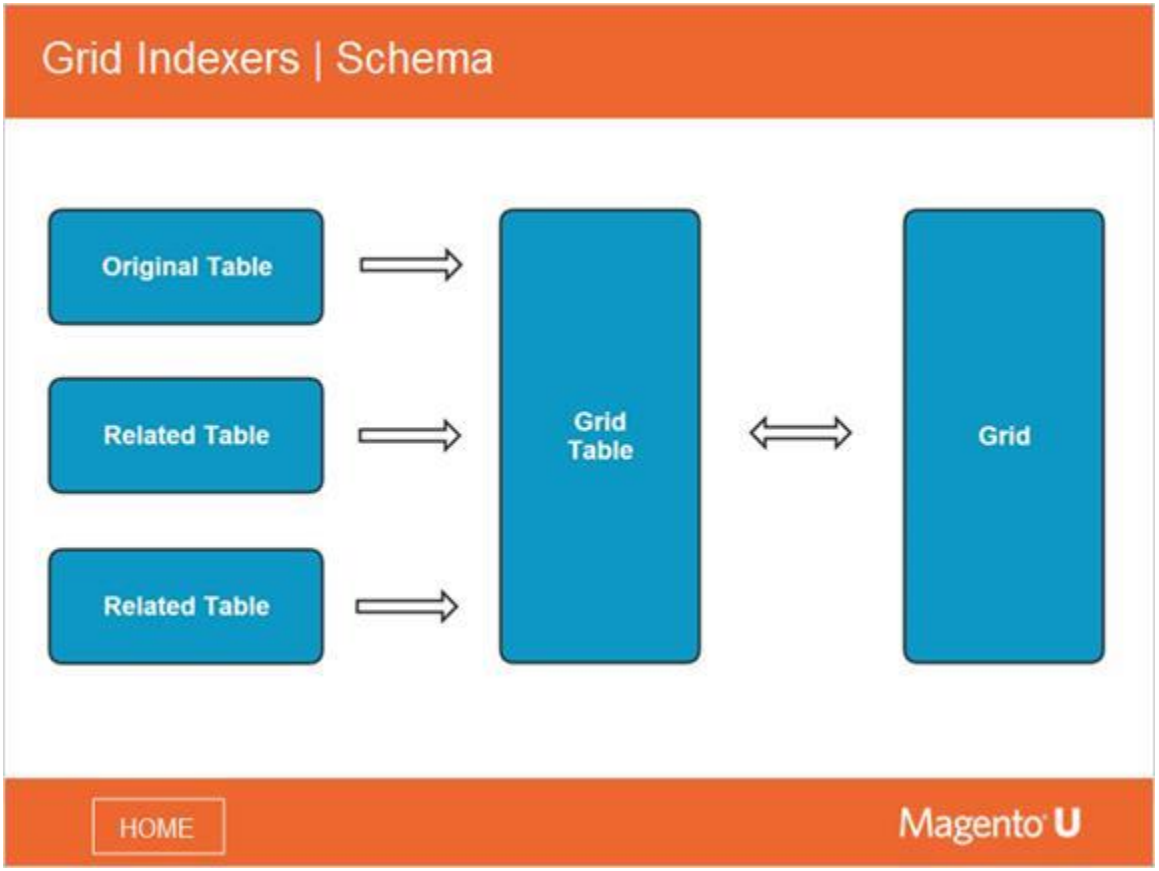


Notes:

In Magento 1, the sales module contains a special set of tables designed for admin grids. These tables populate on the fly, when an entity is saved.

A similar approach is used in Magento 2, but it has been expanded to include more entities.

4.5 Grid Indexers | Schema



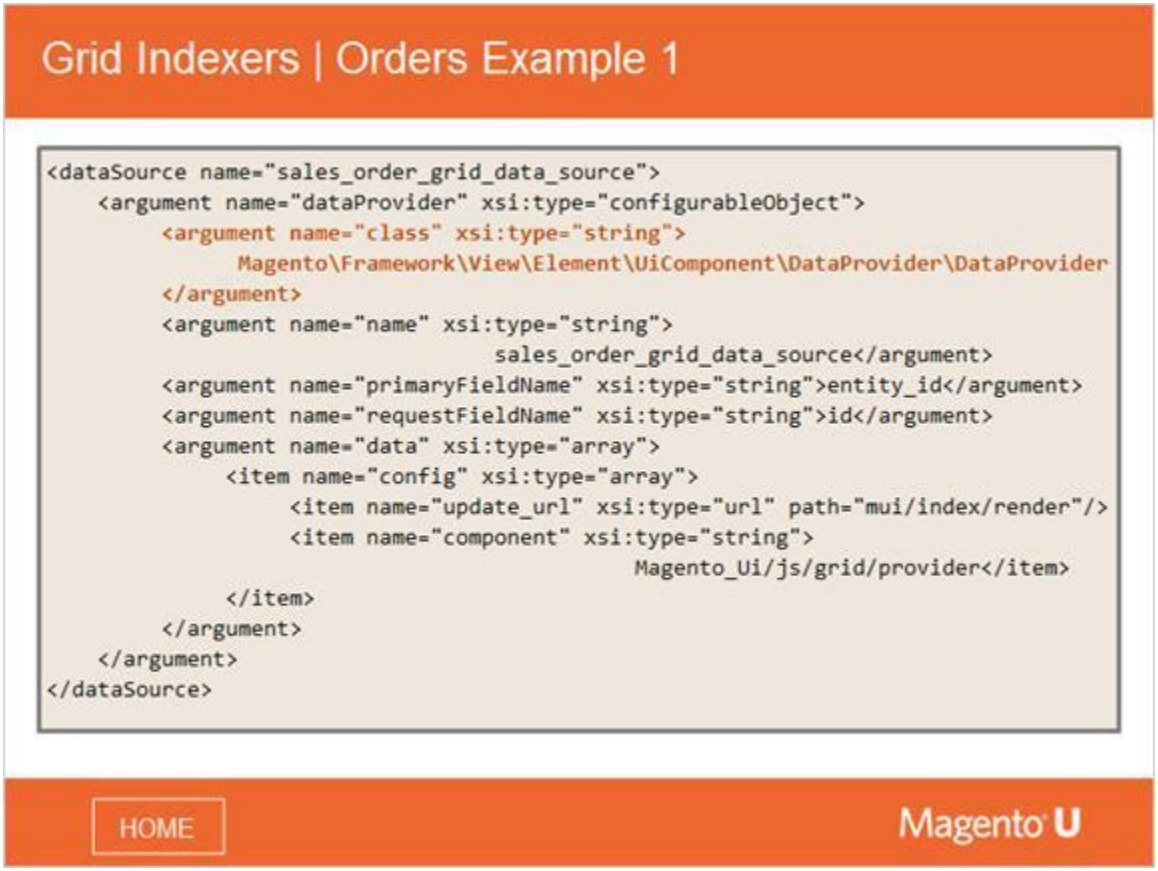
Notes:

Depicted on the slide is a typical schema for grid indexing. Grid indexing solves the problem of joining multiple tables (usually one main table and sets of related ones) in Magento 2.

However, this can result in a large query that doesn't perform well.

A solution is to populate a special “grid” table with the required data at the time of entity saving. This is the same approach used in Magento 1 for sales entities.

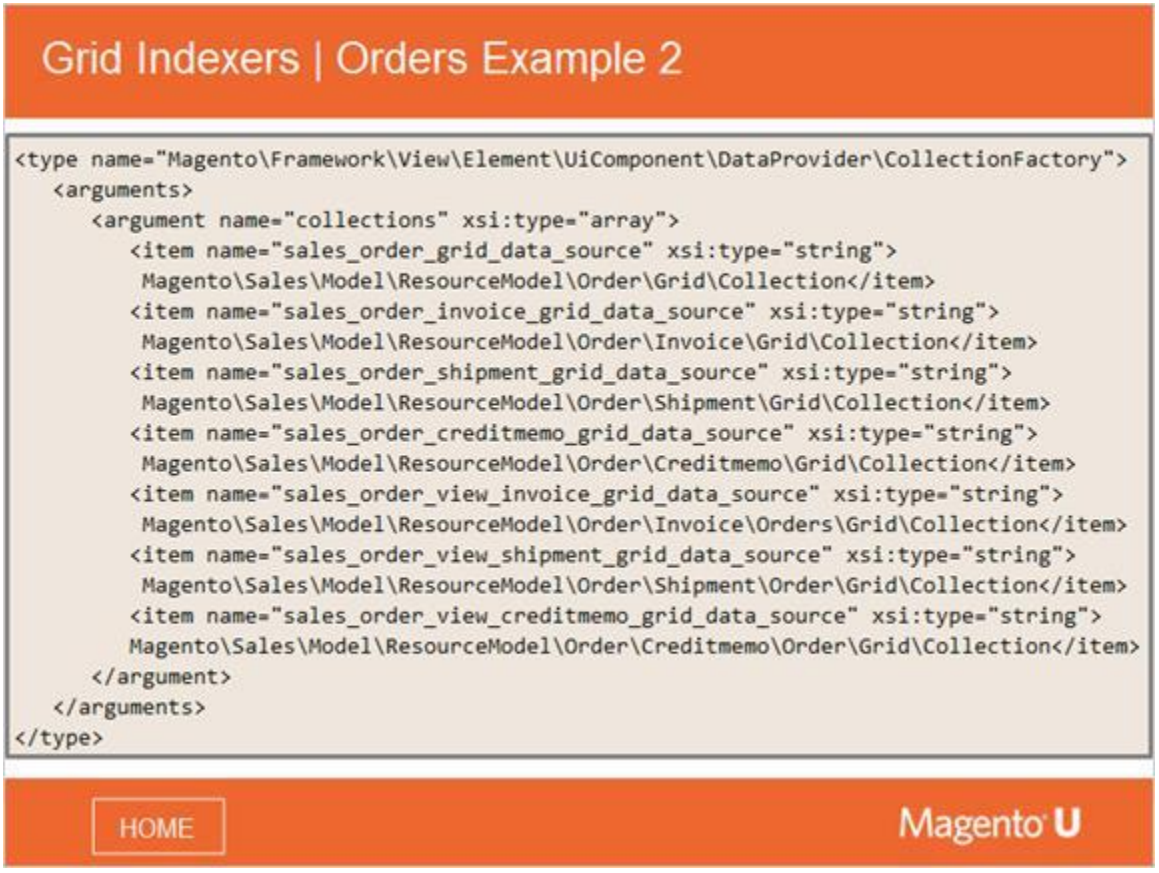
4.6 Grid Indexer | Orders Example 1



Notes:

This is a listing configuration for a sales order. As you can see, it uses an abstract data provider (from the framework, not from the `Magento_Ui` module), which is configured in the `di.xml` (as shown on the next slide).

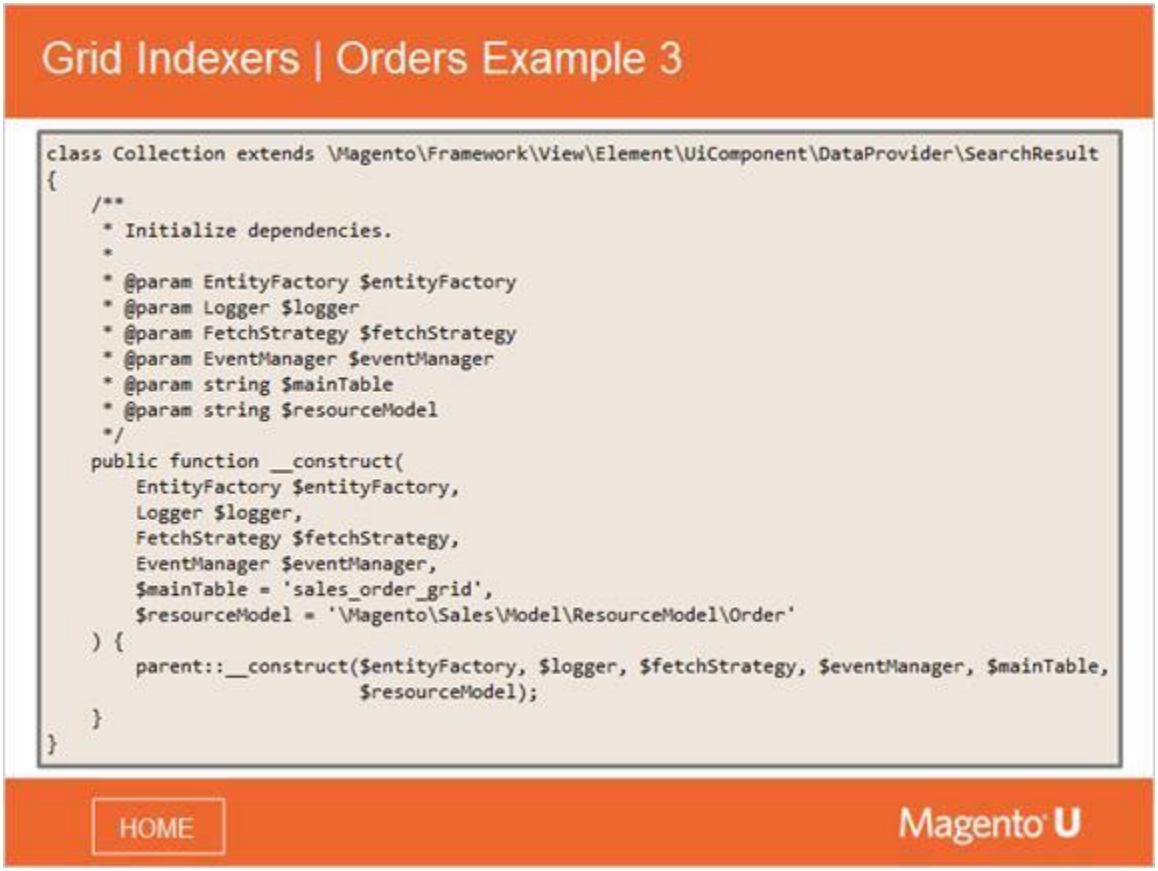
4.7 Grid Indexer | Orders Example 2



Notes:

This configuration from the sales module `di.xml` file shows how grid tables are implemented for sales entities. As you can see, it attaches special collection classes to data sources related to the sales entities.

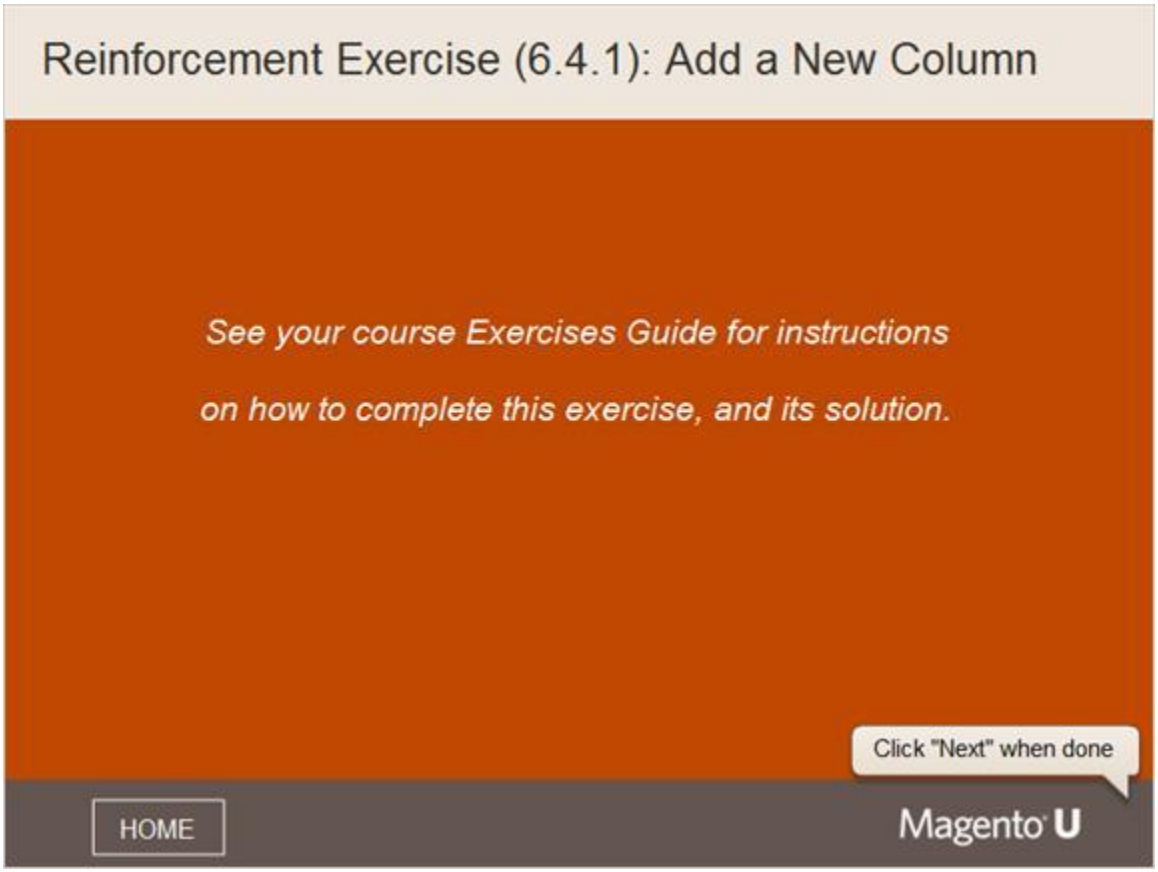
4.8 Grid Indexer | Orders Example 3



Notes:

And finally, looking at a grid collection, you can see it takes data from a special table: `sales_order_grid`.

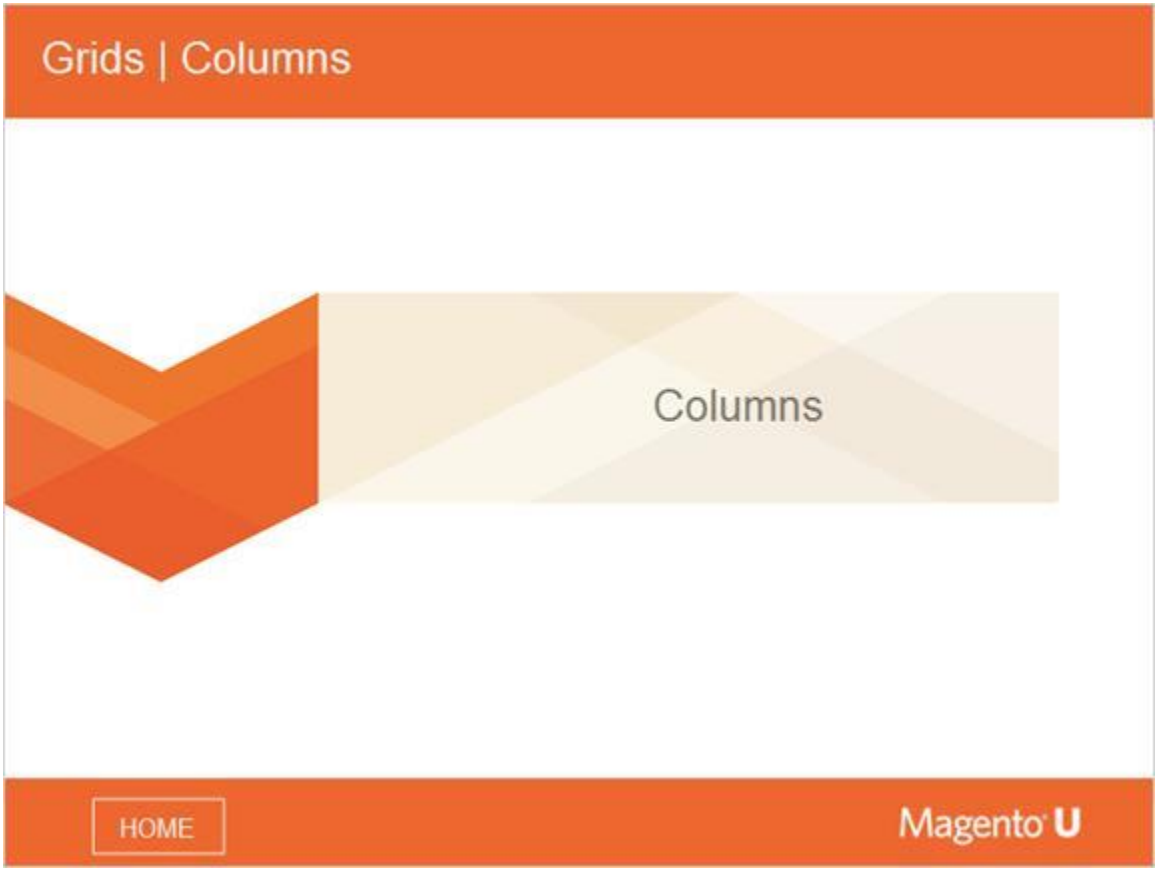
4.9 Reinforcement Exercise (6.4.1)



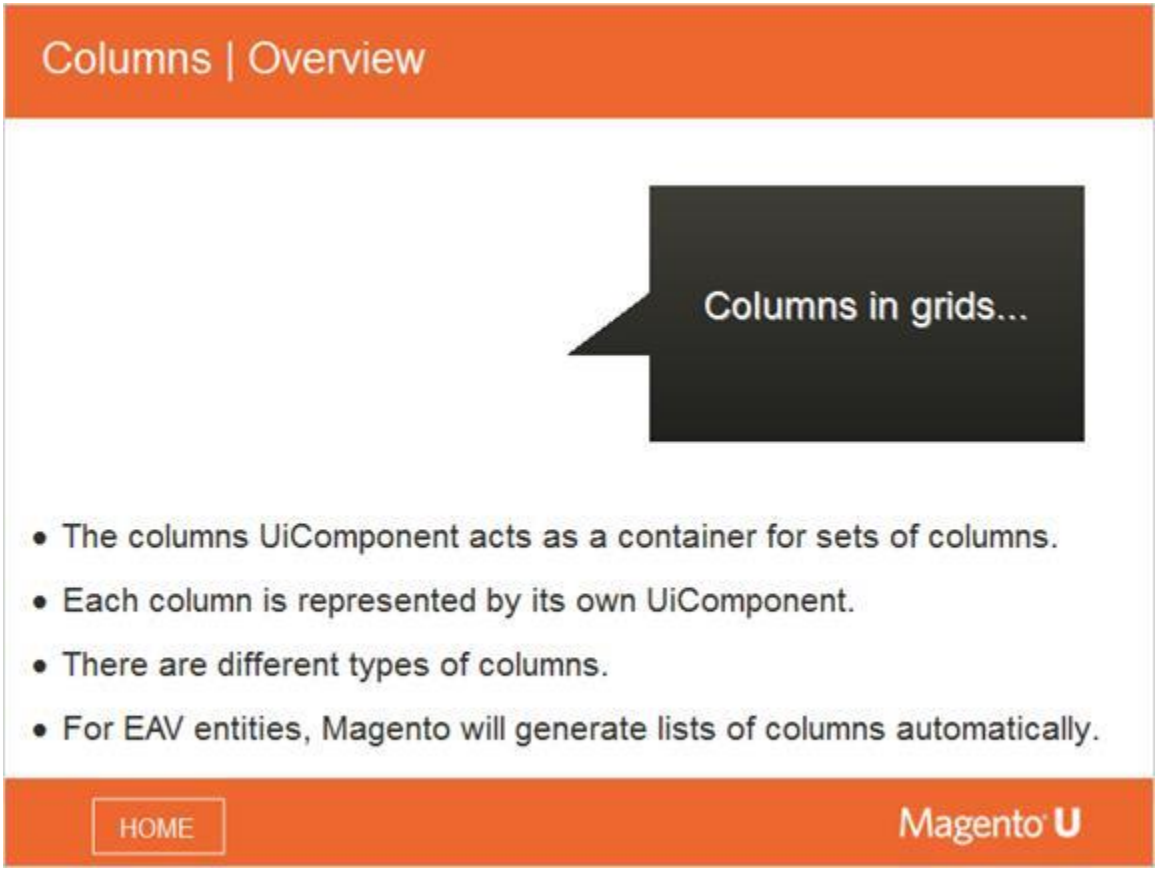
Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

4.10 Grids | Columns



4.11 Columns | Overview



Notes:

For rendering, the columns listing component uses the same approach as with filters.

There is a `Columns` UiComponent that acts as a container, and a set of specific `Column` UiComponents which render specific columns.

In Magento 1, you have to configure columns in the `_prepareColumns()` method of a grid object. In Magento 2, each column is a UiComponent, so it is configured in the component's XML file.

It is important that EAV entities obtain information about columns automatically, although you can see column configuration in the `product_listing.xml` file.

4.12 Columns | definitions.xml

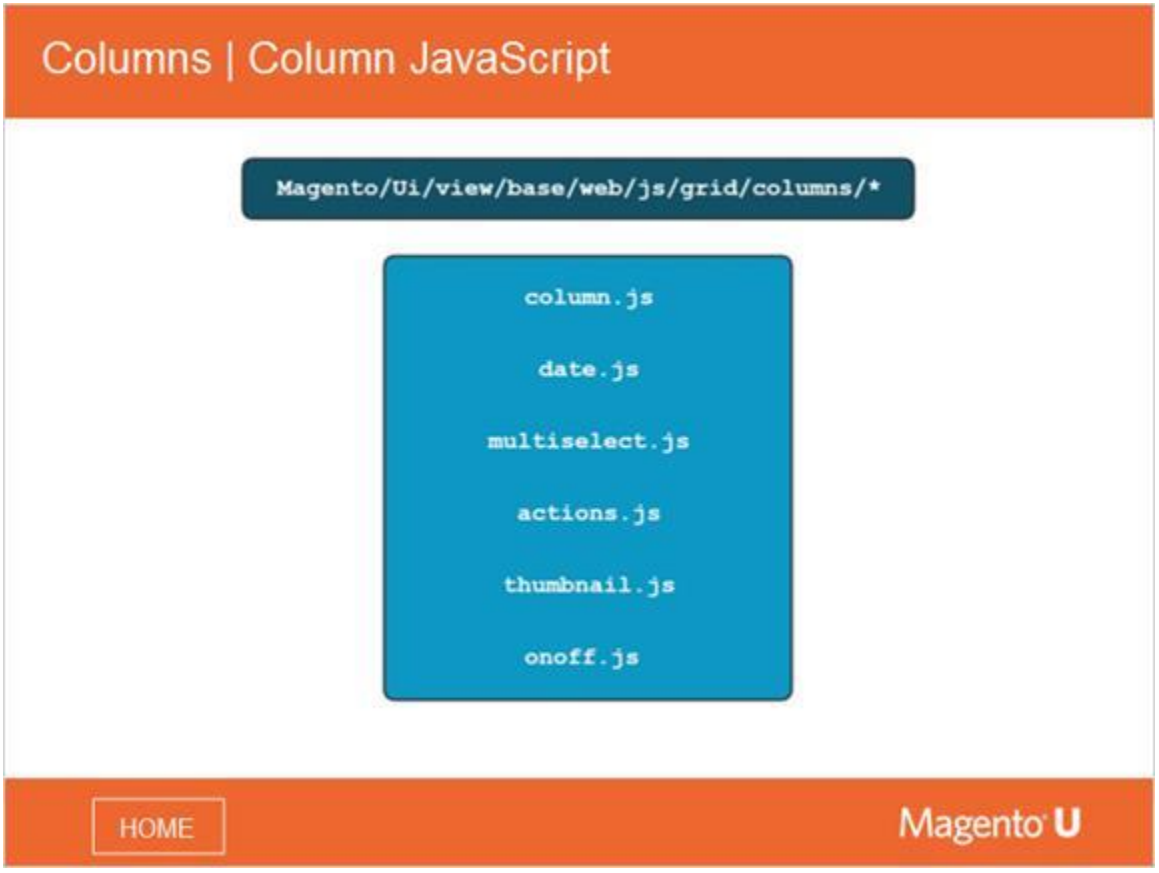

```
<columns class="Magento\Ui\Component\Listing\Columns">
    <argument name="data" xsi:type="array">
        <item name="config" xsi:type="array">
            <item name="component" xsi:type="string">
                                    Magento_Ui/js/grid/listing</item>
            <item name="componentType" xsi:type="string">columns</item>
            <item name="storageConfig" xsi:type="array">
                <item name="provider" xsi:type="string">
                                    ns = ${ $.ns }, index = bookmarks</item>
                <item name="namespace" xsi:type="string">current</item>
            </item>
             ...

        </item>
    </argument>
</columns>

<column class="Magento\Ui\Component\Listing\Columns\Column">
    <argument name="data" xsi:type="array">
        <item name="state_prefix" xsi:type="string">columns</item>
        <item name="config" xsi:type="array">
            <item name="component"
xsi:type="string">Magento_Ui/js/grid/columns/column</item>
            <item name="componentType" xsi:type="string">column</item>
            <item name="dataType" xsi:type="string">text</item>
        </item>
    </argument>
</column
```

Notes:

The `columns` UiComponent is defined in the `definition.xml` file.

In the above code, you can see the columns component with the corresponding class `Magento\Ui\Component\Listing\Columns` (first highlighted code) and the `column` component with the class `Magento\Ui\Component\Listing\Columns\Column` (second highlight).

4.13 Columns | Column JavaScript



Notes:

This slide lists JavaScript modules that are responsible for rendering columns.

4.14 Columns | column.js

Notes:

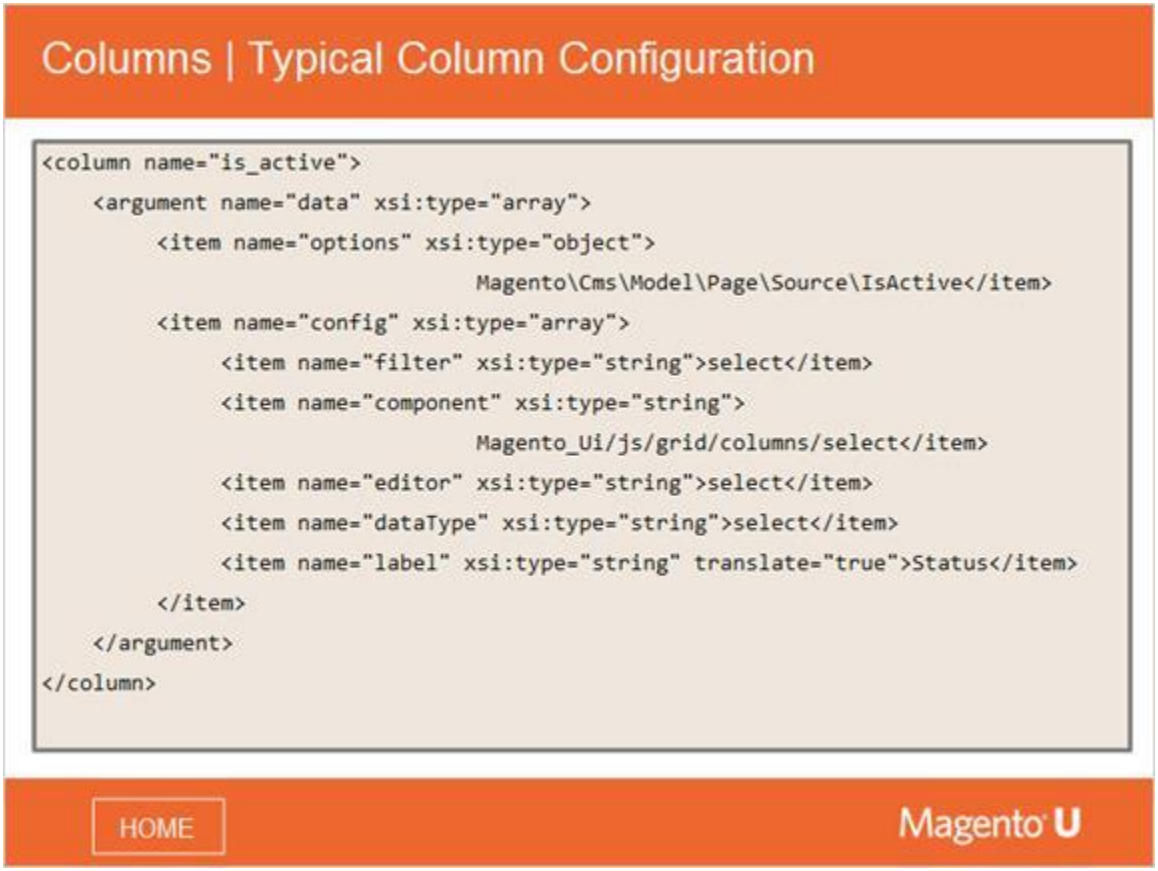
`column.js` is a JavaScript module that renders a column. If you open the file to take a look at the constructor, you will see the following code.

This code provides information about which configuration options are available in the listing xml files:

```
return Element.extend({
        defaults: {
            headerTmpl: 'ui/grid/columns/text',
            bodyTmpl: 'ui/grid/cells/text',
            disableAction: false,
            controlVisibility: true,
            sortable: true,
            sorting: false,
            visible: true,
            draggable: true,
            fieldClass: {},
            ignoreTmpls: {
                fieldAction: true
            },
            statefull: {
                visible: true,
```

```
            sorting: true
        },
        imports: {
            exportSorting: 'sorting'
        },
        listens: {
            '${ $.provider }:params.sorting.field': 'onSortChange'
        },
        modules: {
            source: '${ $.provider }'
        }
    }
```

4.15 Columns | definitions.xml



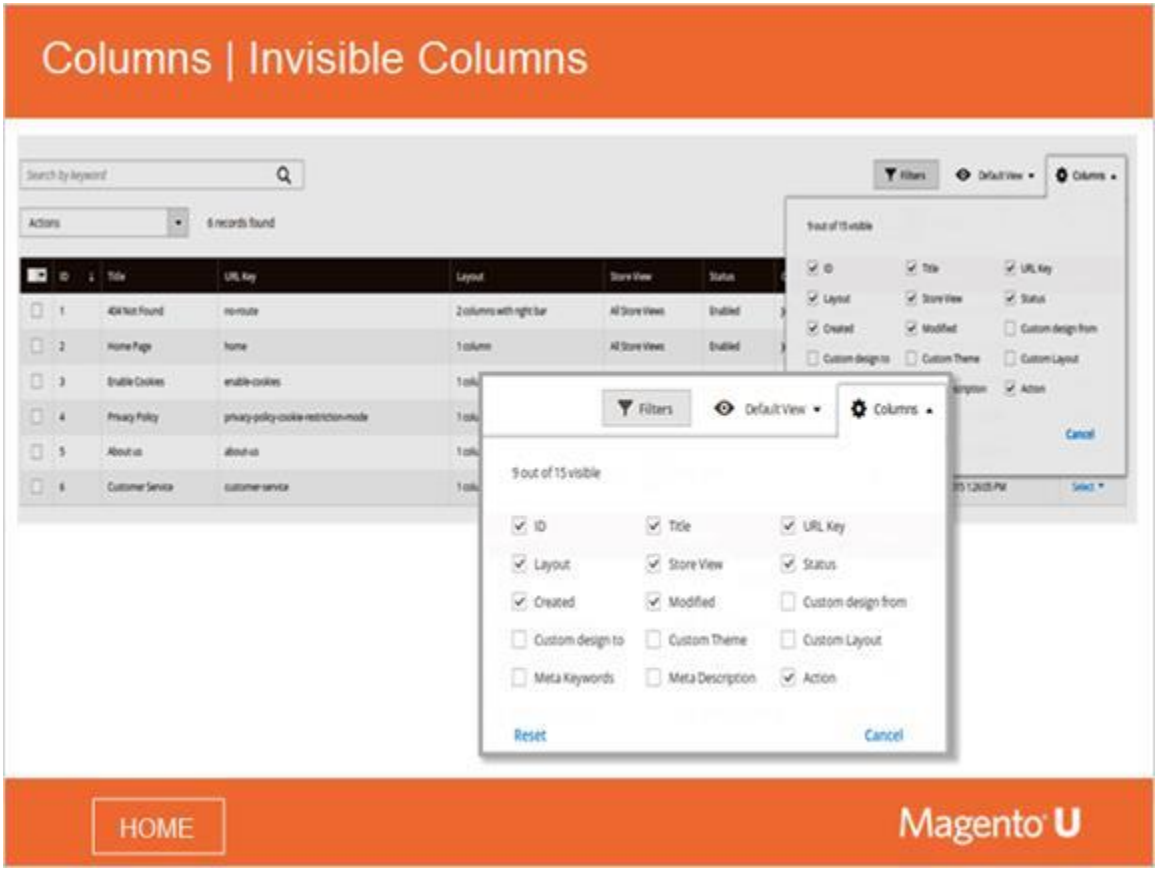
Notes:

This code displays an example of column configuration in the Cms module (Cms Page grid).

This column is of the type `select`, so you can specify options. The options are set in the `Magento\Cms\Model\Page\Source\IsActive` class.

You can also see that the component that renders this column is `select.js`.

4.16 Columns | Invisible Columns



Notes:

Magento 2 offers a unique feature not available in Magento 1: dynamic columns. This is possible because of the flexibility that JavaScript rendering provides.

As you can see on the slide, there is a “Columns” button that opens a list of all the available columns. Columns with a check mark are those currently rendered; the remaining columns are those that can be rendered later.

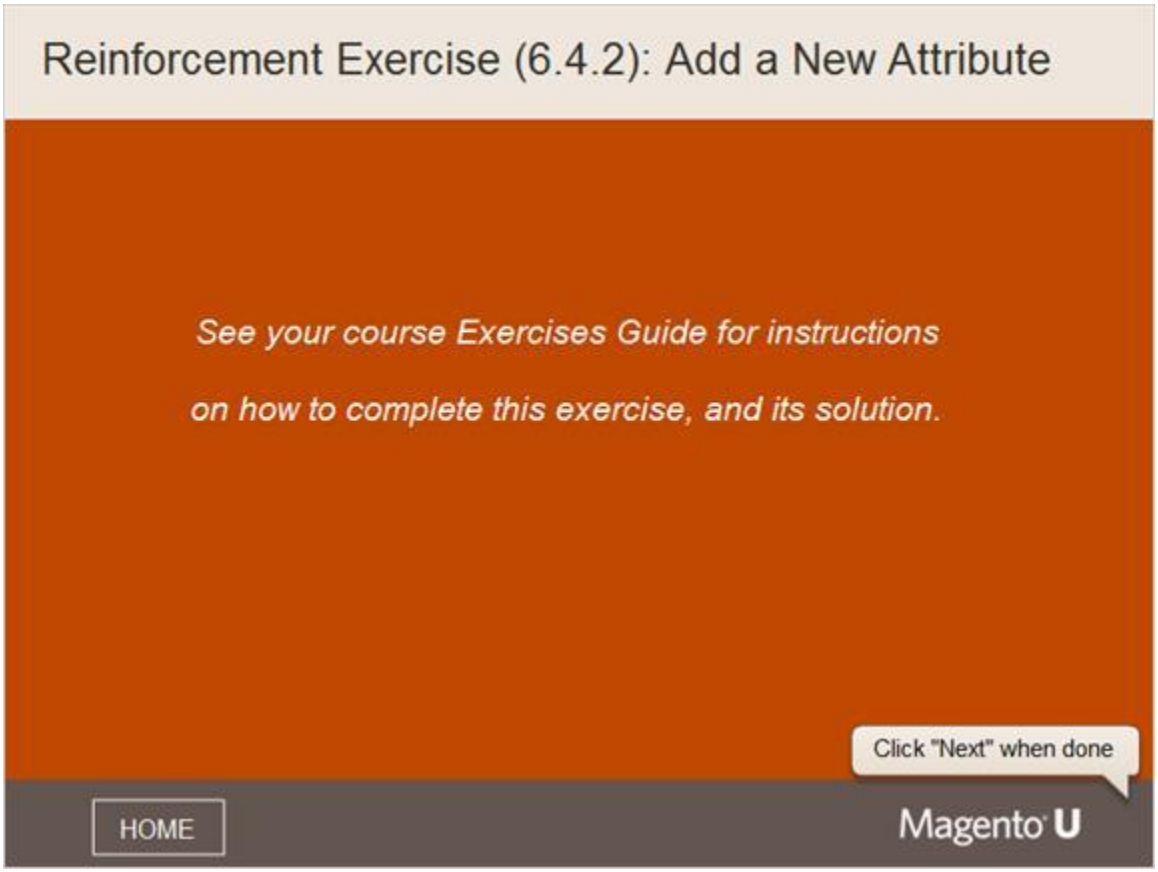
There is also an option in the column configuration (Boolean visible) that defines whether a column is rendered by default or not. The next slide illustrates this.

4.17 Columns | Invisible Column Configuration

Notes:

Here is the code example for a non-visible column configuration.

4.18 Reinforcement Exercise (6.4.2)



Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

4.19 Grids | Mass Actions

Notes:

We now move to a discussion of mass actions.

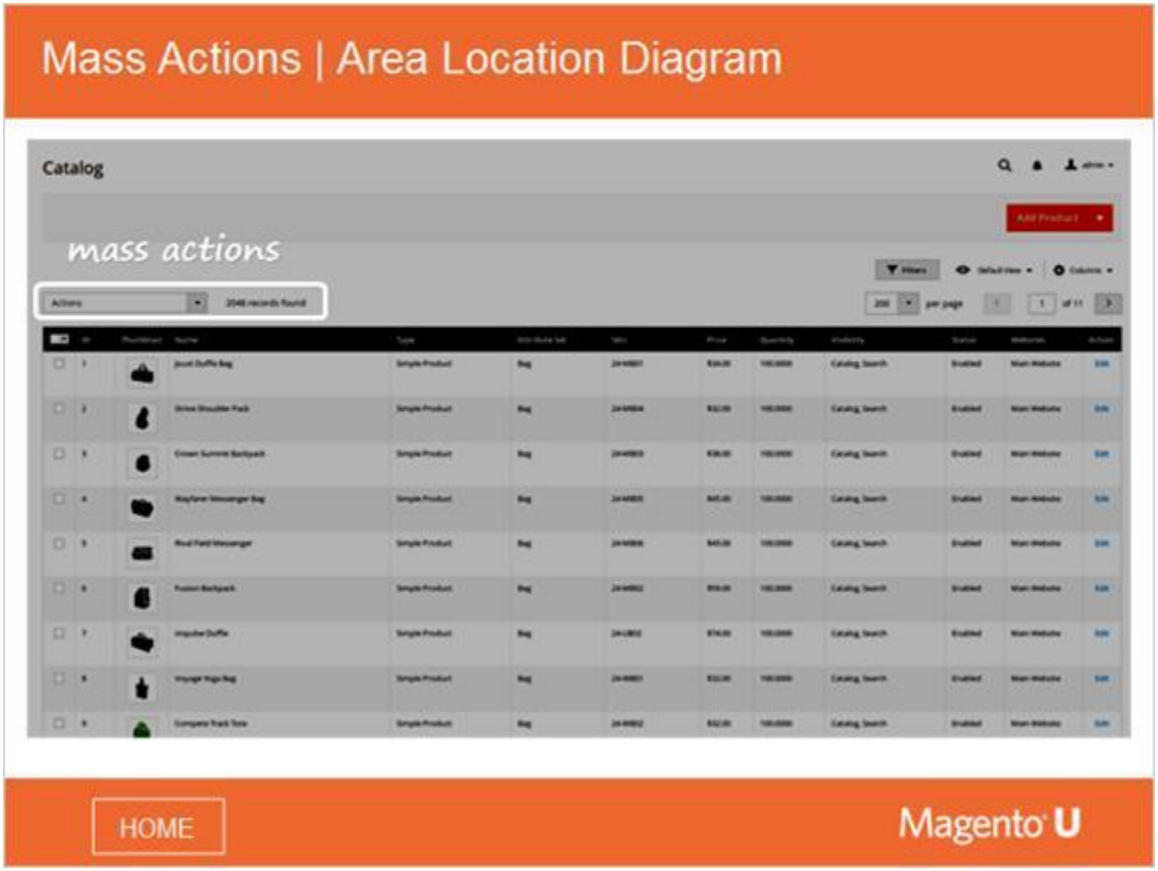
4.20 Mass Actions | Overview

Notes:

Mass actions are used to apply certain actions (events) to a set of records.

They define which actions are available on the grid page, and list actions configured in the `data/config/actions` argument.

4.21 Mass Actions | Area Location Diagram



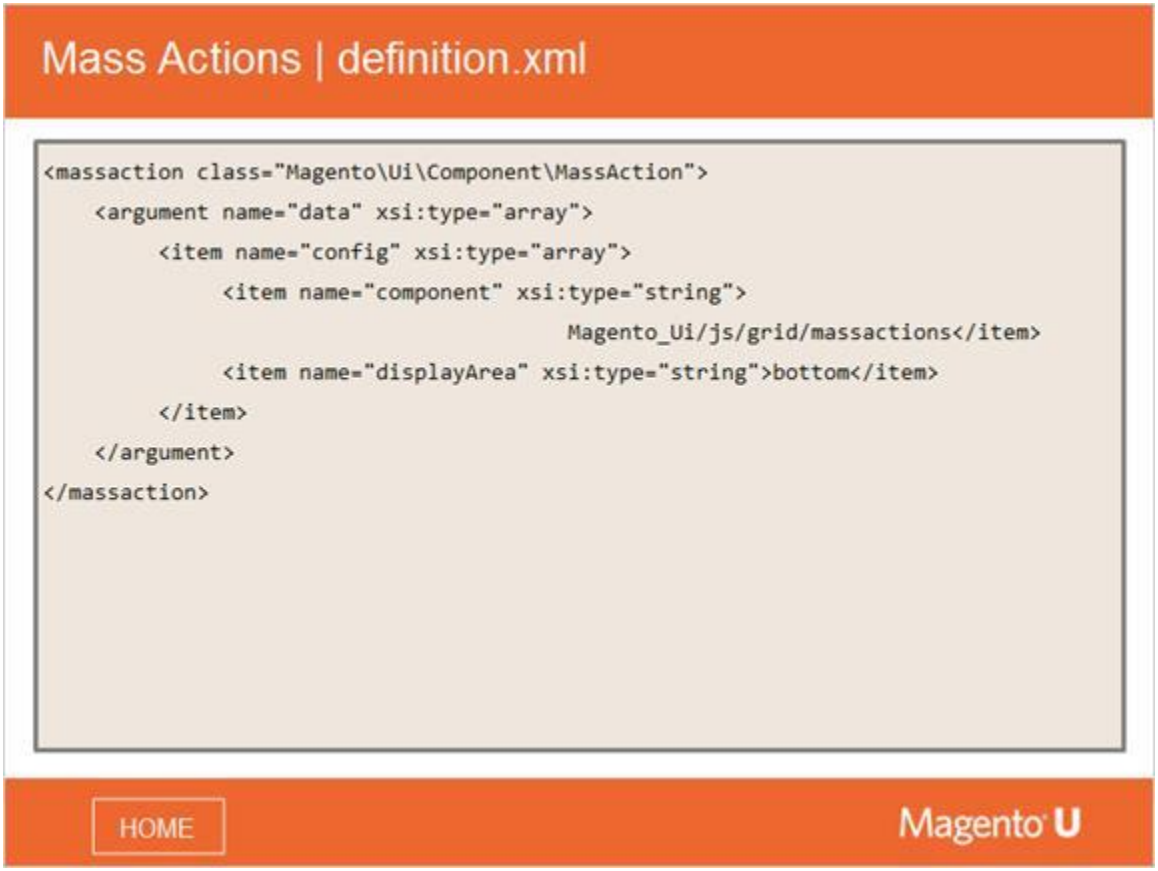
Notes:

Recall from the general Area Location diagram where the Massactions block is situated.

The massactions dropdown can optionally be made available on a grid page. If so, it defines a list of actions that can be applied to a select set of records from the grid.

Conceptually, massactions is a UiComponent, child of a Listing.

4.22 Mass Actions | definitions.xml



Notes:

In this code example, you can see the generic configuration for `massactions` defined in `definition.xml`.

In that file, you can see the corresponding UiComponent class - `Magento\Ui\Compnent\MassAction`, the template - `templates/listingcontainer/massaction/default.xhtml`, and the JavaScript module responsible for rendering - `massactions.js`.

4.23 Mass Actions | Actions Config Example

Notes:

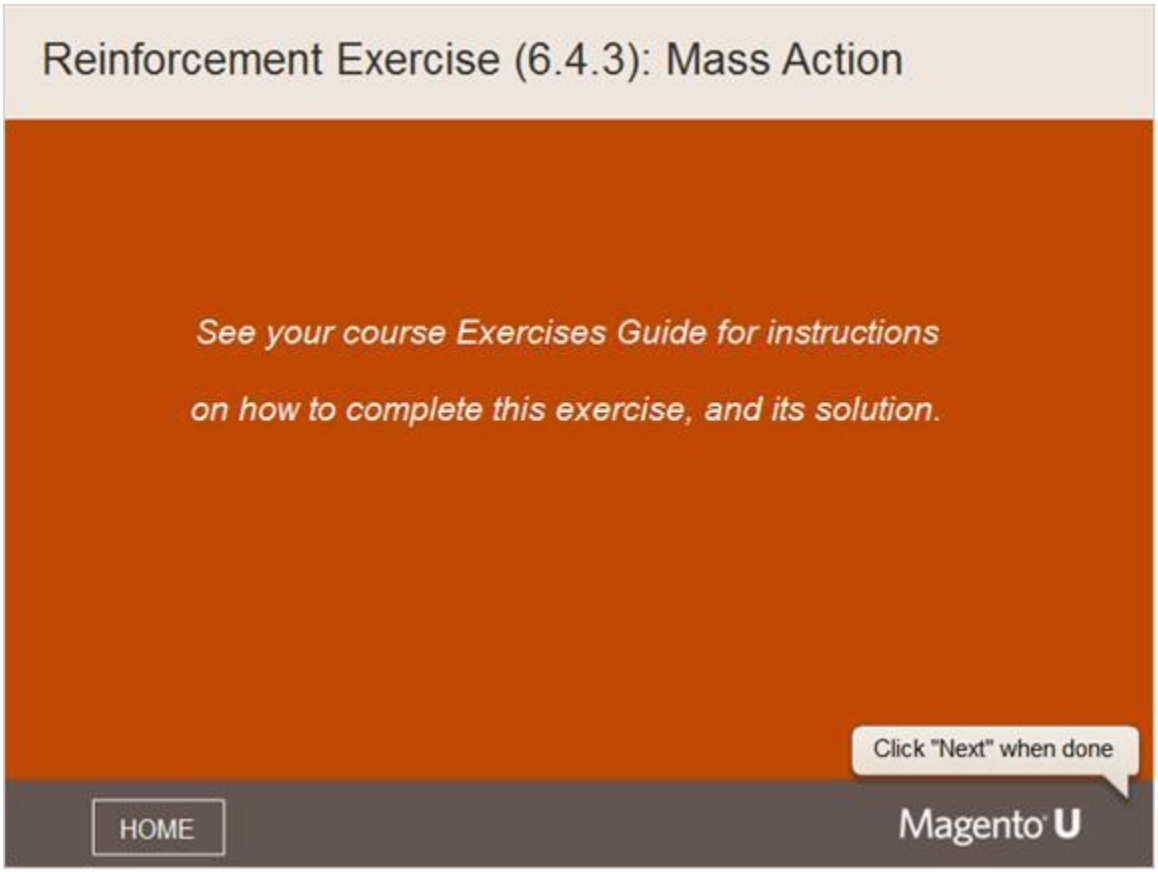
Here is the configuration from a Cms page grid. You see the possible actions -- `Delete, Disable` -- along with the corresponding URLs to send data, and possible confirmation messages.

4.24 Mass Actions | JavaScript Module

Notes:

The JavaScript module responsible for `massactions` rendering is `massactions.js`.

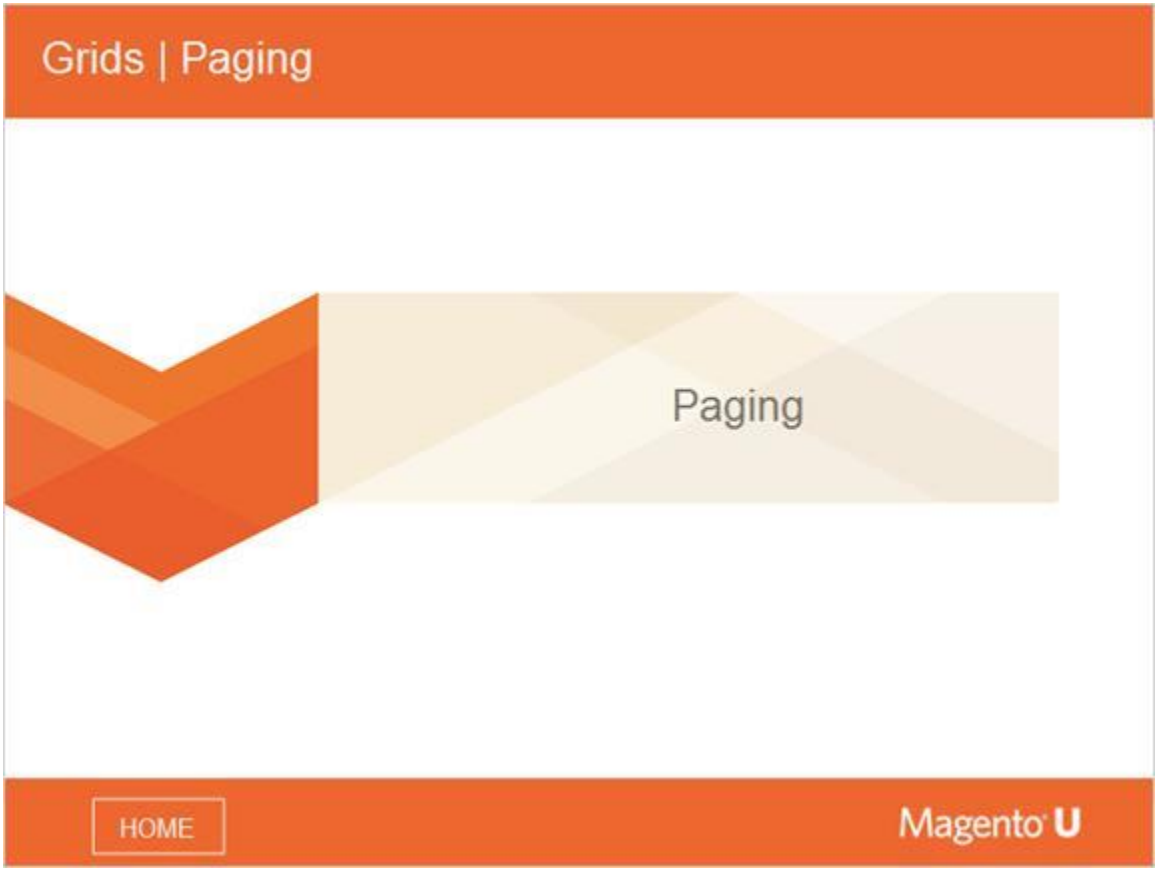
4.25 Reinforcement Exercise (6.4.3)



Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

4.26 Grids | Paging



Notes:

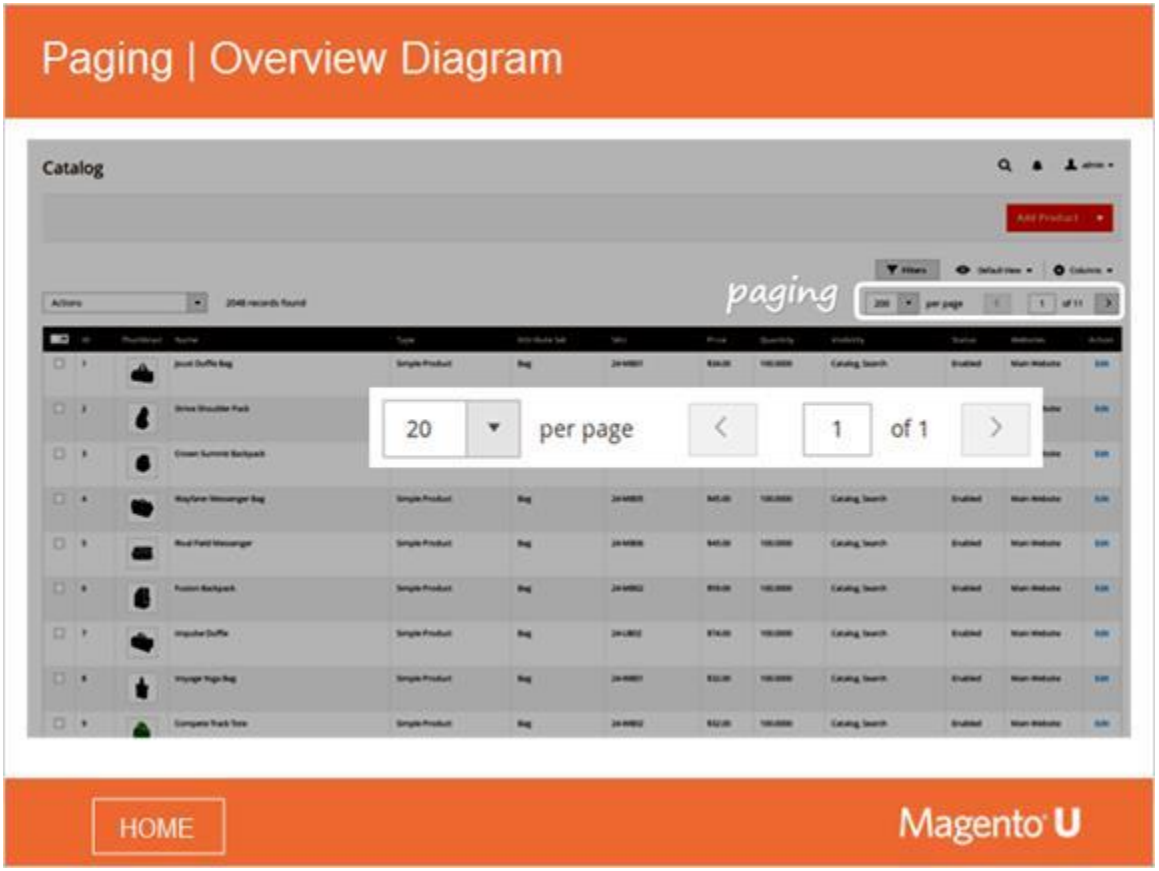
Now, we'll discuss the final topic in this module, Paging.

4.27 Paging | Overview

Notes:

The paging component is used to render paging elements:
- Next / previous pages
- Current page
- Total number of records
- Set of closest next / previous page numbers

4.28 Paging | Overview Diagram



Notes:

Again using the general Area Location diagram, you can see where the Paging feature is situated.

Paging is also a UiComponent. Its main responsibility is to display paging control options, like number of records per page, page number, and page navigation.

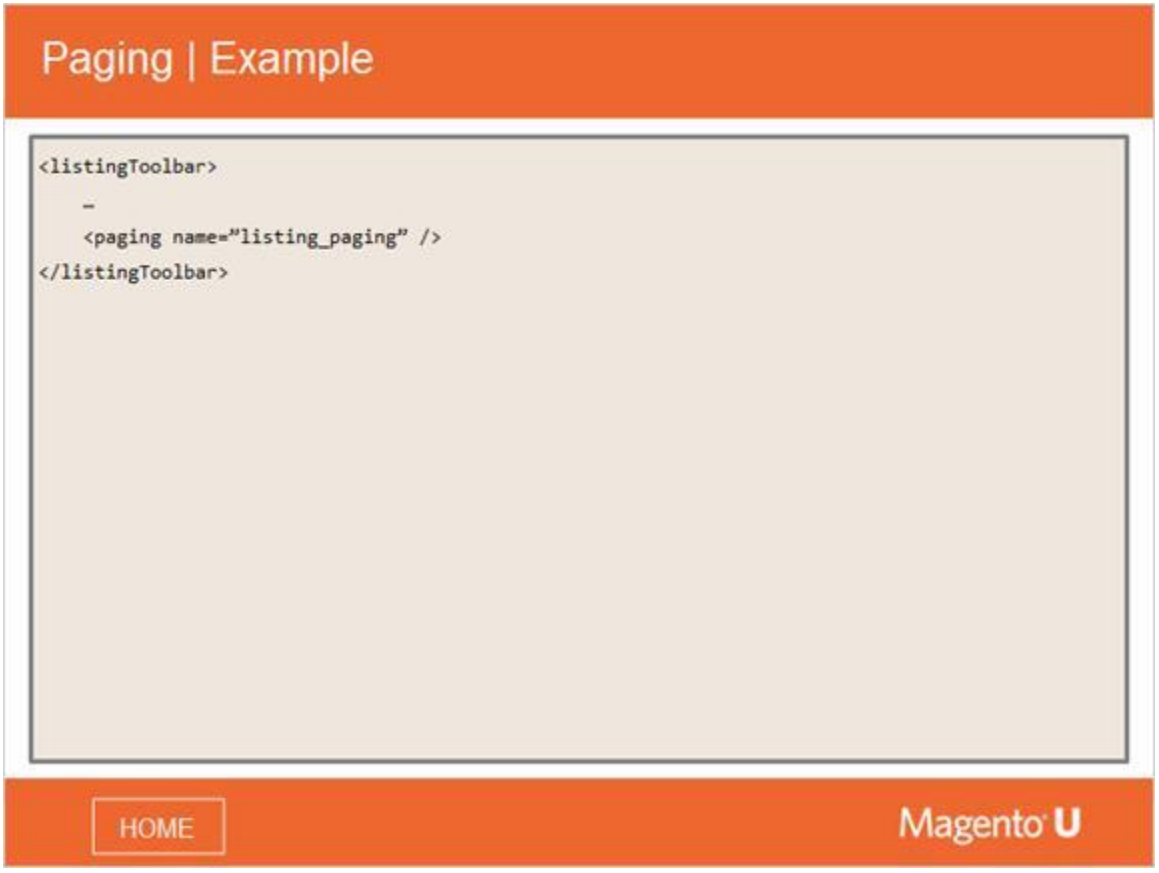
4.29 Paging | default.xml

Notes:

The code example shows paging configuration in the file, `definition.xml`. As you can see, it is quite simple.

Note that the JavaScript module is `paging.js`, located under the `Magento/Ui` module.

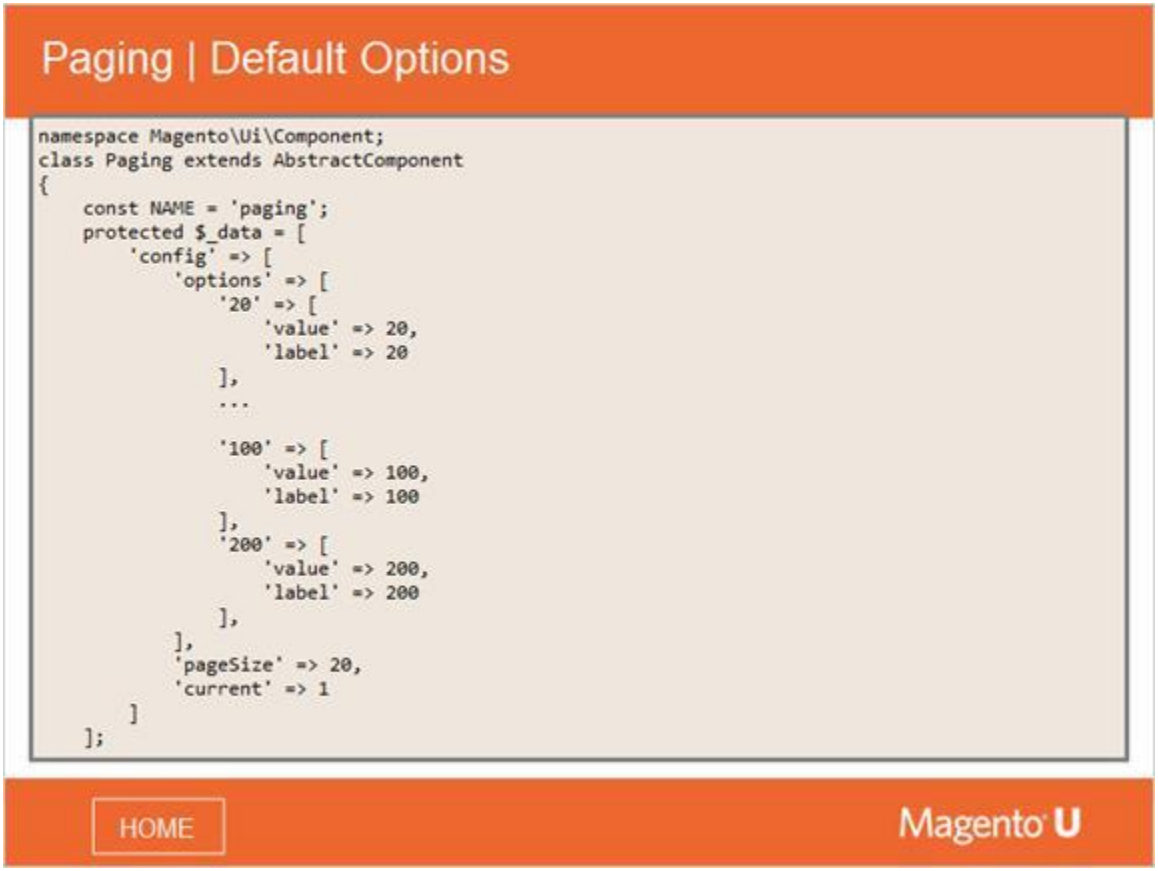
4.30 Paging | Example



Notes:

This slide shows an example of paging configuration for the product grid.

4.31 Paging | Default Options



Notes:

Here you can see how the default paging options are configured in its PHP UiComponent class.

4.32 Reinforcement Exercise (6.4.4)

Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

5. Forms

5.1 Adminhtml Forms

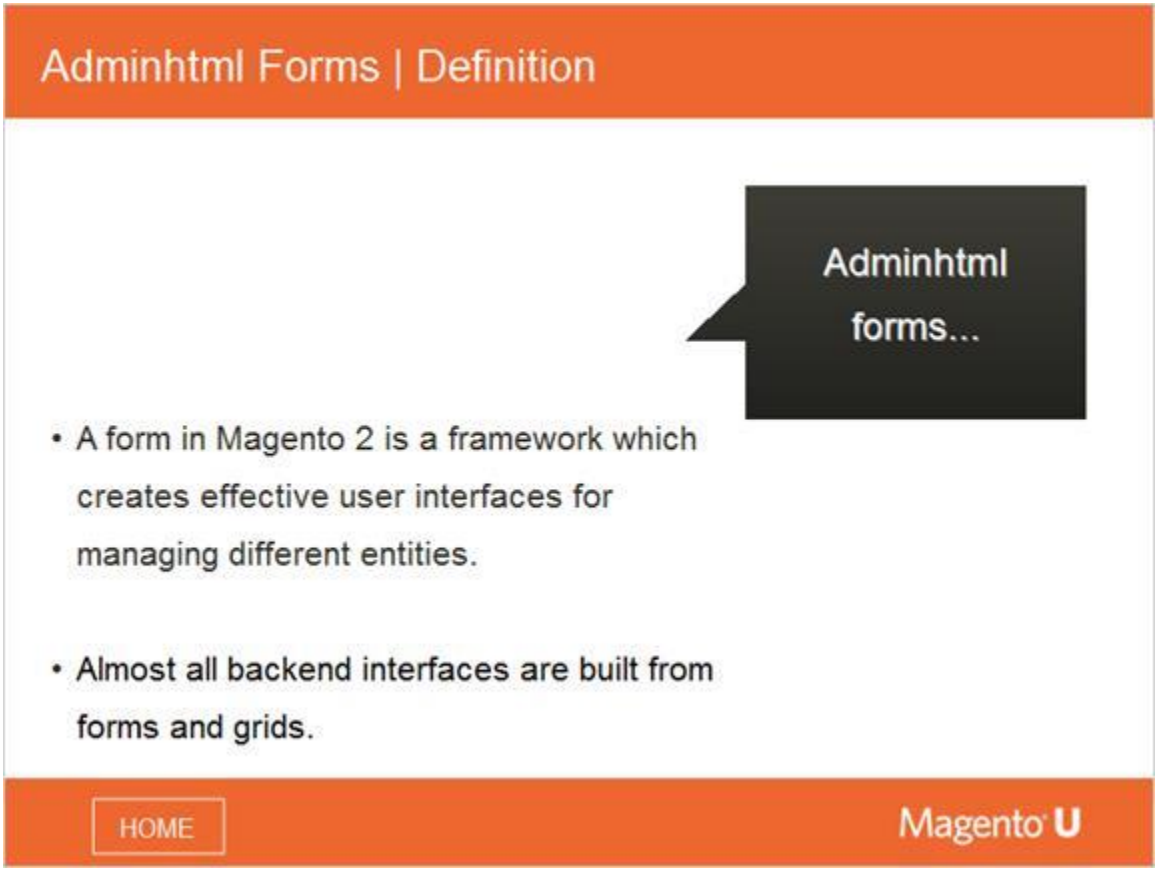
Notes:

5.2 Module Topics | Adminhtml Forms

Notes:

In this module, we will discuss:
- Forms overview
- Form components
- Fieldsets
- Form elements

5.3 Adminhtml Forms | Definition



Notes:

A form in Magento 2 is a framework, which creates effective user interfaces for managing different entities.

Almost all backend interfaces are built from forms and grids.

5.4 Adminhtml Forms | Format & Fields

Notes:

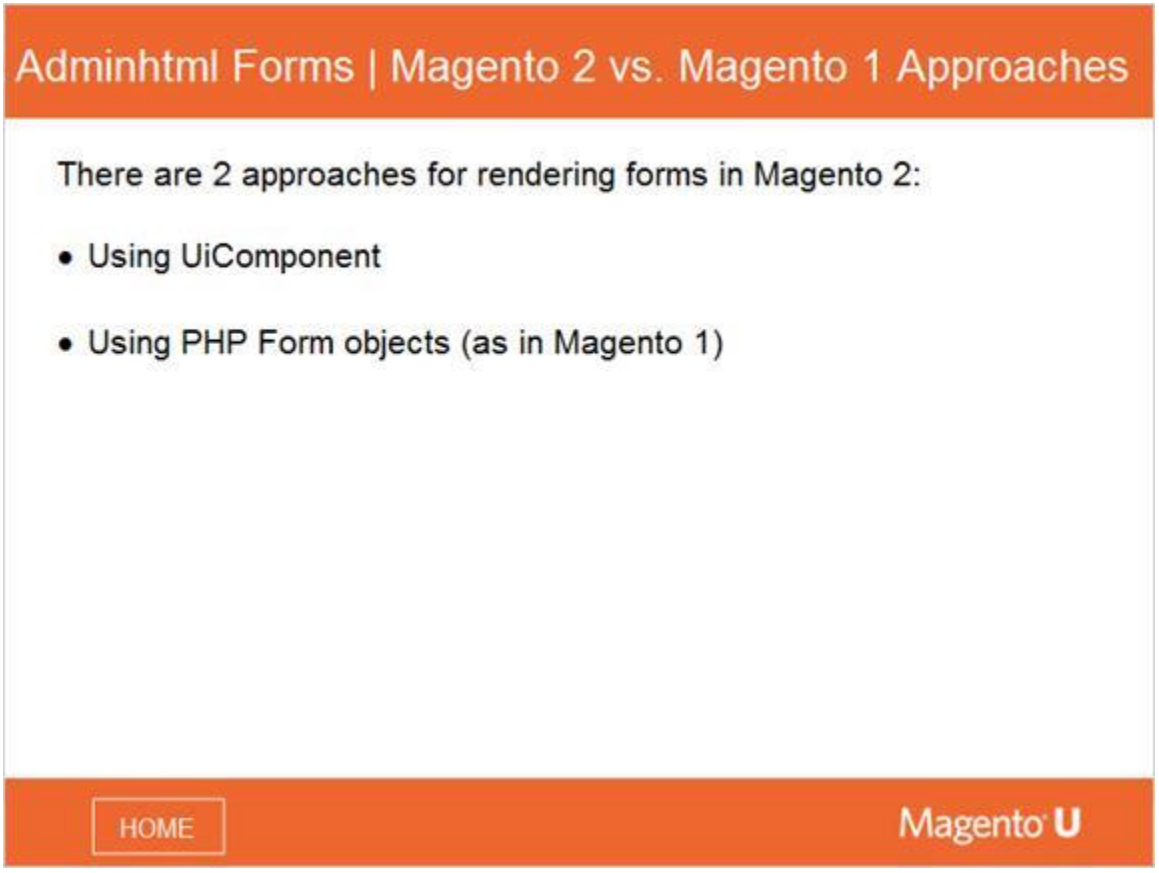
Here is an example of the form backend interface.

5.5 Adminhtml Forms | Form Container

Notes:

We will start by looking at a form container. Its main goal is to display actions applicable to the whole form -- for example, Save.

5.6 Adminhtml Forms | Magento 2 vs. Magento 1 Approaches



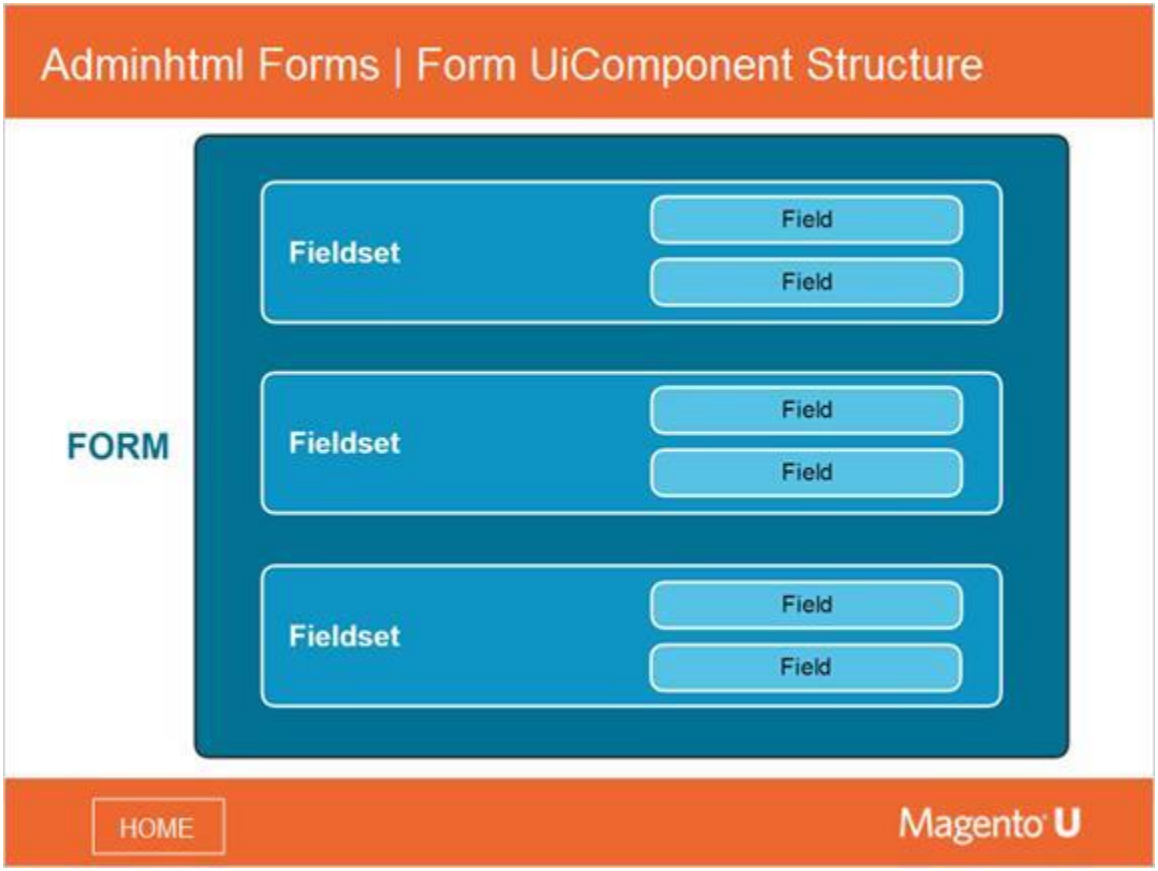
Notes:

In Magento 1, forms are generated using a standard layout mechanism.

In Magento 2, there are two options: using a set of PHP classes (blocks), or the `UiComponent`.

In this course, we will cover the `UiComponent` approach.

5.7 Adminhtml Forms | Form UiComponent Structure



Notes:

The Form UiComponent's PHP class is: `Magento\Ui\Component\Form`

Its most important function is to obtain data:

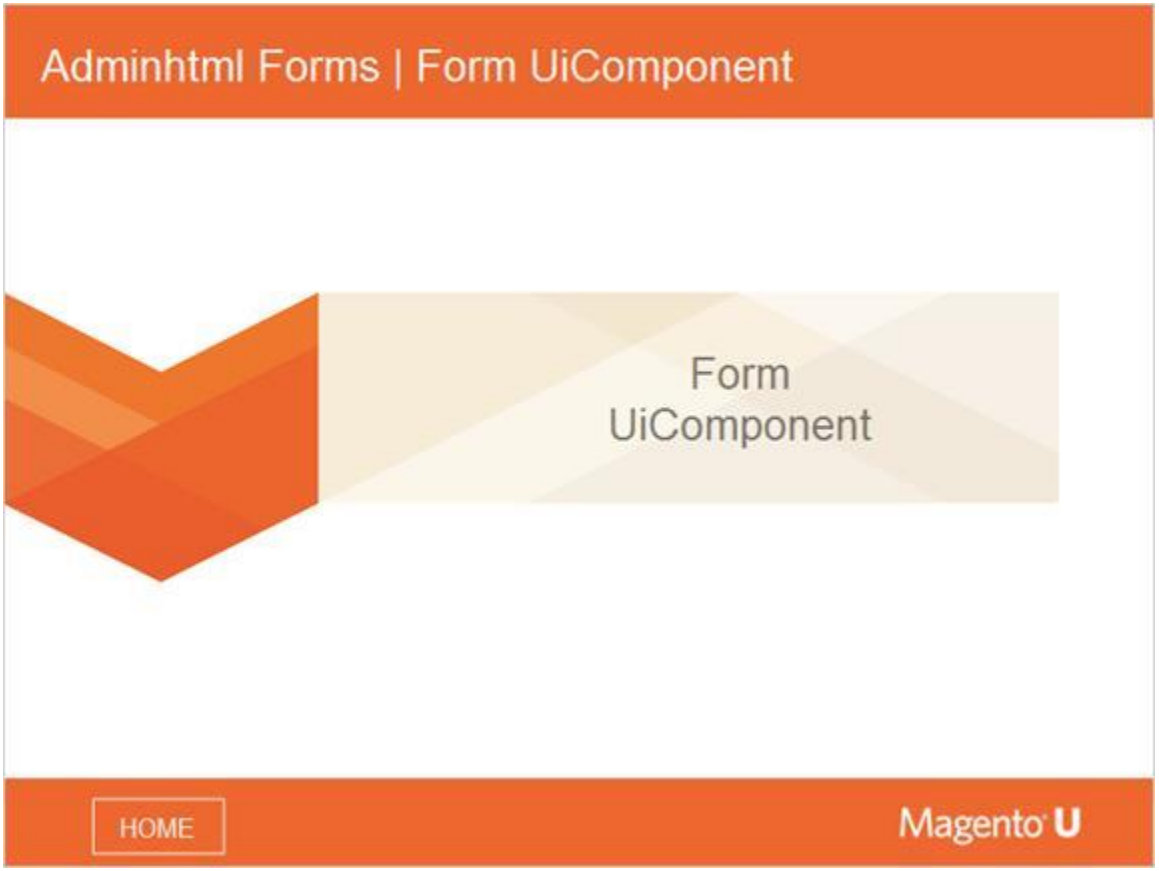
```
public function getDataSourceData()
    {
        $dataSource = [];

        $id = $this->getContext()->getRequestParam($this->getContext()->getDataProvider()-
>getRequestFieldName(), null);
        $filter = $this->filterBuilder->setField($this->getContext()->getDataProvider()-
>getPrimaryFieldName())
            ->setValue($id)
            ->create();
        $this->getContext()->getDataProvider()
            ->addFilter($filter);

        $data = $this->getContext()->getDataProvider()->getData();

        if (isset($data[$id])) {
            $dataSource = [
```



```
                'data' => $data[$id]
            ];
        } elseif (isset($data['items'])) {
            foreach ($data['items'] as $item) {
                if ($item[$item['id_field_name']] == $id) {
                    $dataSource = ['data' => ['general' => $item]];
                }
            }
        }
        return $dataSource;
    }
```

As you can see, it checks whether the "id" parameter is set, and if so, adds a primary key filter.

5.8 Adminhtml Forms | Form UiComponent



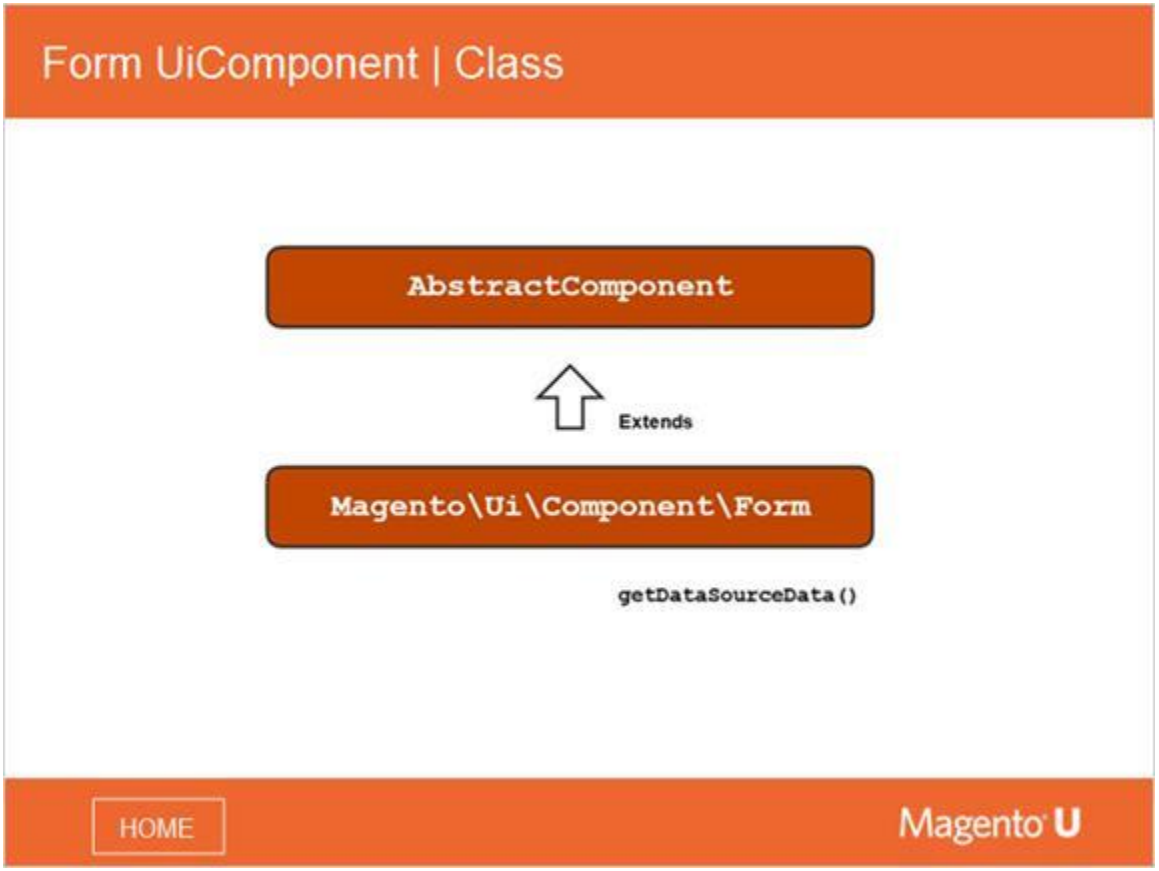
5.9 Form UiComponent | Code Definition

Notes:

This code shows how the Form component is defined in the file: `definition.xml`.

From this XML code, you can see that the component's code is located in `Magento/Ui/view/base/web/js/form/form.js`, and the default template in: `Magento/Ui/view/base/ui_component/templates/form/default.xhtml`

5.10 Form UiComponent | DataProvider Customer Example



Notes:

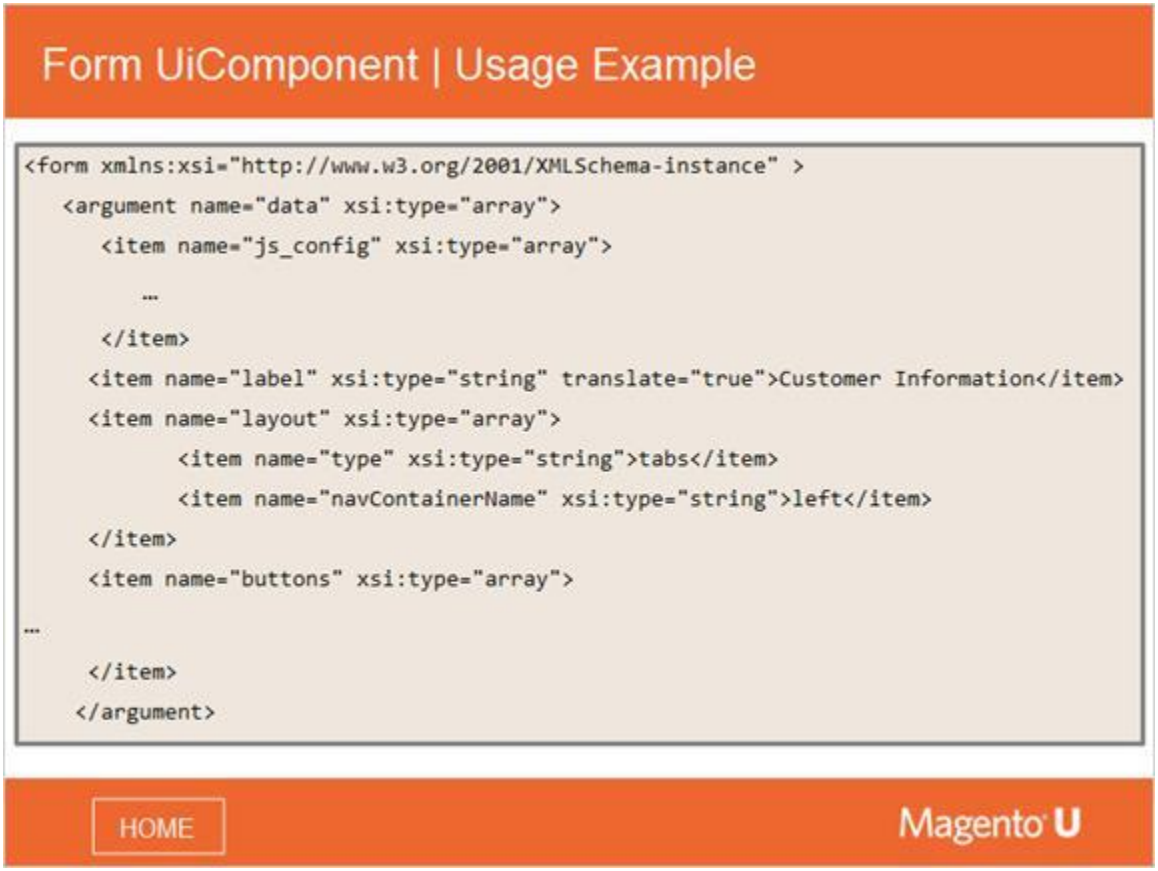
The Form UiComponent's PHP class is: `Magento\Ui\Component\Form`.

Its most important function is to obtain data. As you can see, it checks whether the "`id`" parameter is set, and if so, adds a primary key filter.

```
 public function getDataSourceData()
    {
        $dataSource = [];

        $id = $this->getContext()->getRequestParam($this->getContext()->getDataProvider()-
>getRequestFieldName(), null);
        $filter = $this->filterBuilder->setField($this->getContext()->getDataProvider()-
>getPrimaryFieldName())
            ->setValue($id)
            ->create();
        $this->getContext()->getDataProvider()
            ->addFilter($filter);

        $data = $this->getContext()->getDataProvider()->getData();

        if (isset($data[$id])) {
```

```
            $dataSource = [
                'data' => $data[$id]
            ];
        } elseif (isset($data['items'])) {
            foreach ($data['items'] as $item) {
                if ($item[$item['id_field_name']] == $id) {
                    $dataSource = ['data' => ['general' => $item]];
                }
            }
        }
        return $dataSource;
    }
```

5.11 Form UiComponent | Usage Example



Notes:

This is an example from the Customer module of how the form UiComponent can be used.

Some things to note:
- `js_config:` Standard field for a UiComponent
- `label:` Self-explanatory
- `layout` Standard UiComponent directive
- `buttons:` This item includes button setup for a form. (The next slide provides more detail about this.)

5.12 Form UiComponent | Magento Block - Buttons

Notes:

This example demonstrates how to configure a list of buttons. Each button is defined by an item element; its corresponding value is a Magento block that represents the required button.

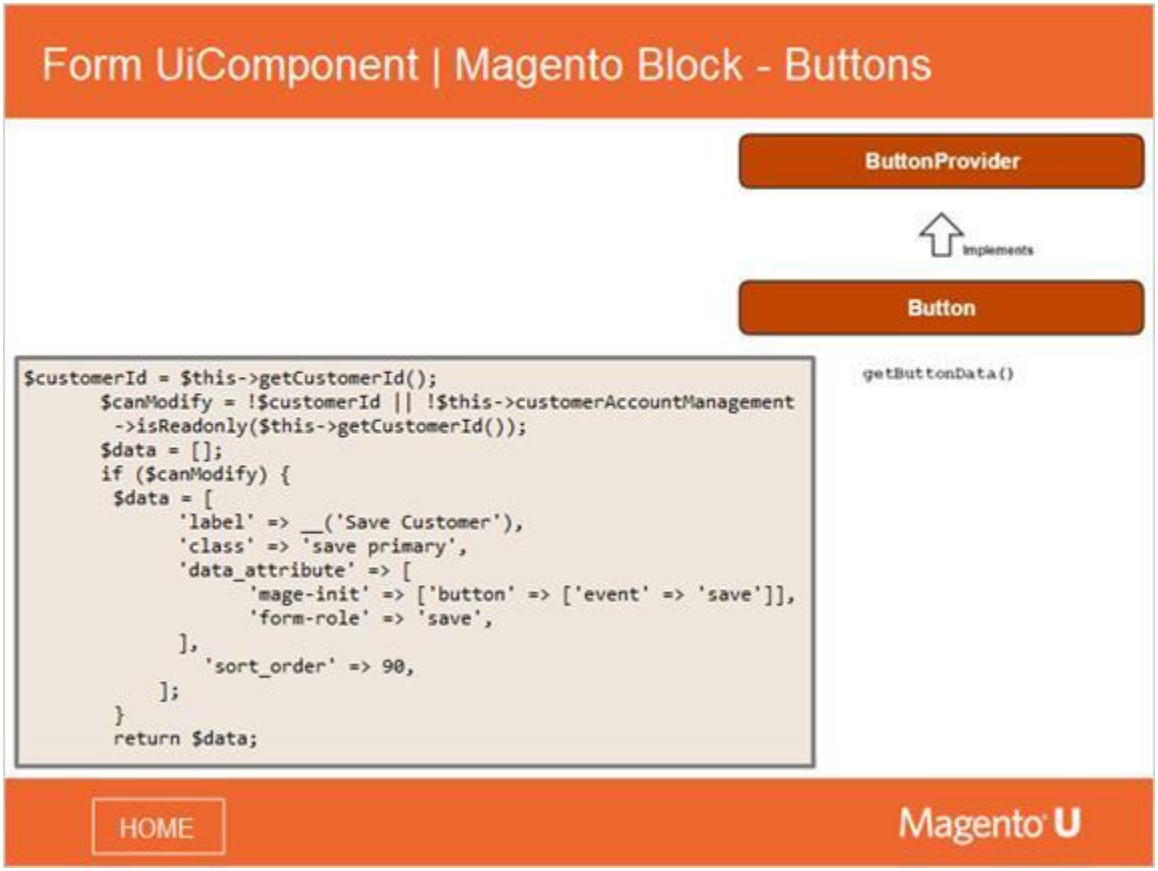
The second code example, for the `BackButton`, represents a typical button example. As you analyze the code, you will see that `getButtonData()` returns an array with button configuration data like `label`, `onClick` handler, css class and `sort_order`.

`getBackUrl` is an example of button-specific functionality. For example, `DeleteButton` includes `getDeleteUrl()`.

```
class BackButton extends GenericButton implements ButtonProviderInterface
{
    /**
     * @return array
     */
    public function getButtonData()
    {
        return [
            'label' => __('Back'),
            'on_click' => sprintf("location.href = '%s';", $this->getBackUrl()),
            'class' => 'back',
            'sort_order' => 10
        ];
```

```
    }

    /**
     * Get URL for back (reset) button
     *
     * @return string
     */
    public function getBackUrl()
    {
        return $this->getUrl('*/*/');
    }
}
```

5.13 Form UiComponent | Magento Block - Buttons

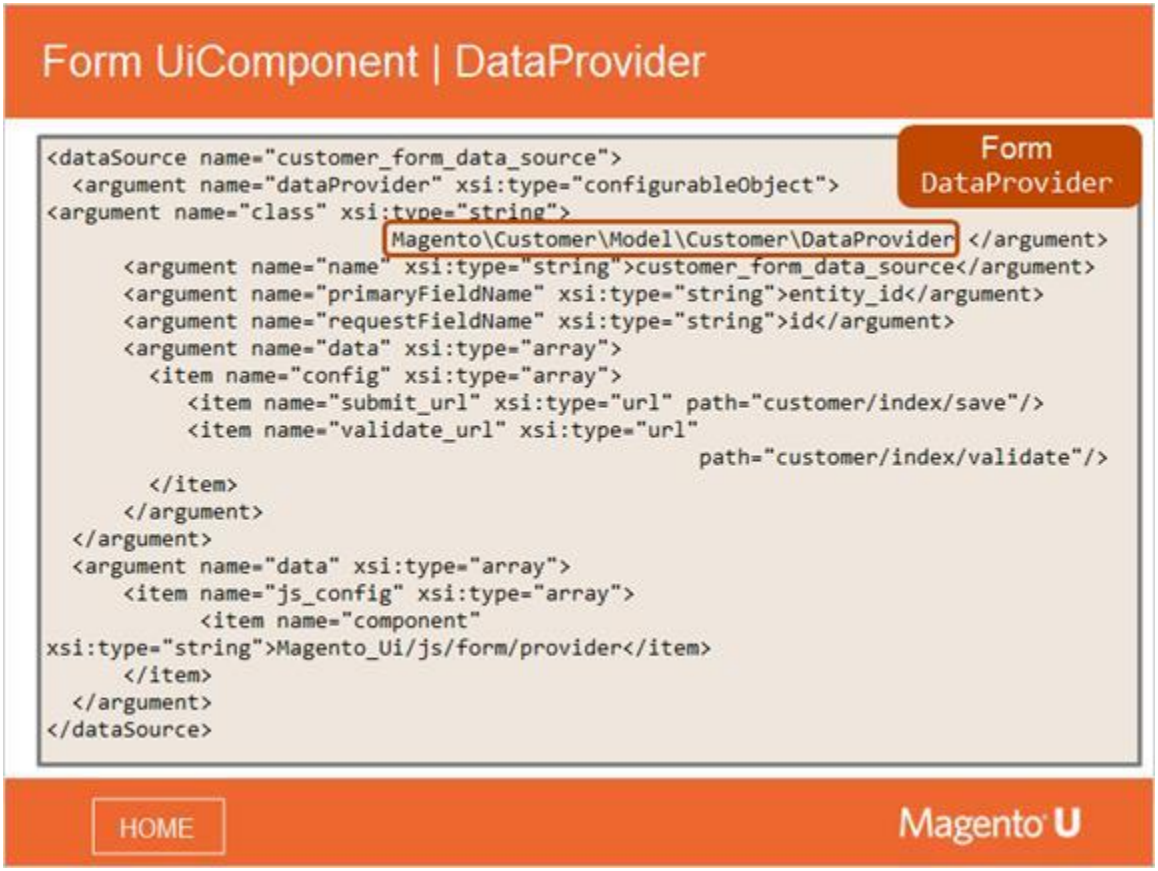


Notes:

Continuing the code example, `Button` implements `ButtonProvider`, which is `Magento\Framework\View\Element\UiComponent\Control\ButtonProviderInterface.`

Note the additional button properties included in the `data_attribute` element of the `getButtonData` result.

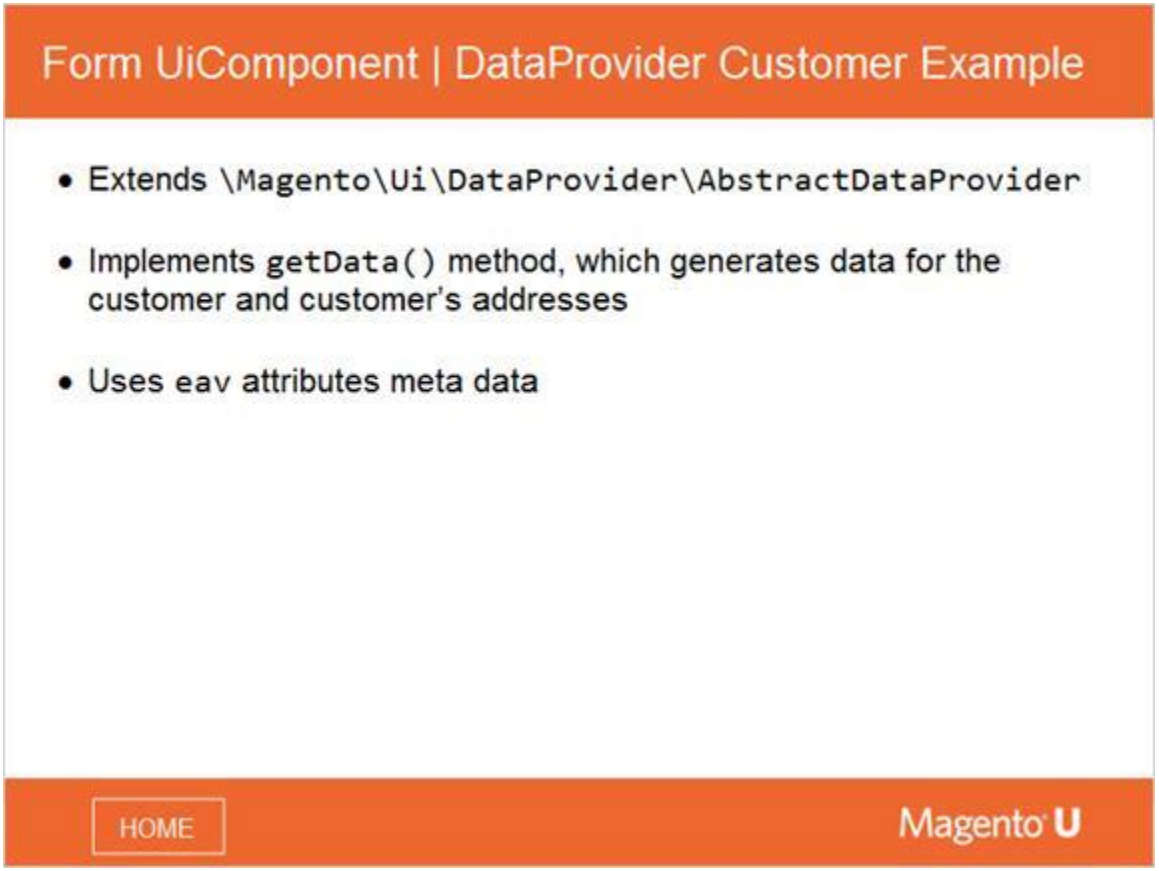
5.14 Form UiComponent | DataProvider



Notes:

Here is a code example for a `DataProvider` definition for a customer form.

5.15 Form UiComponent | DataProvider Customer Example

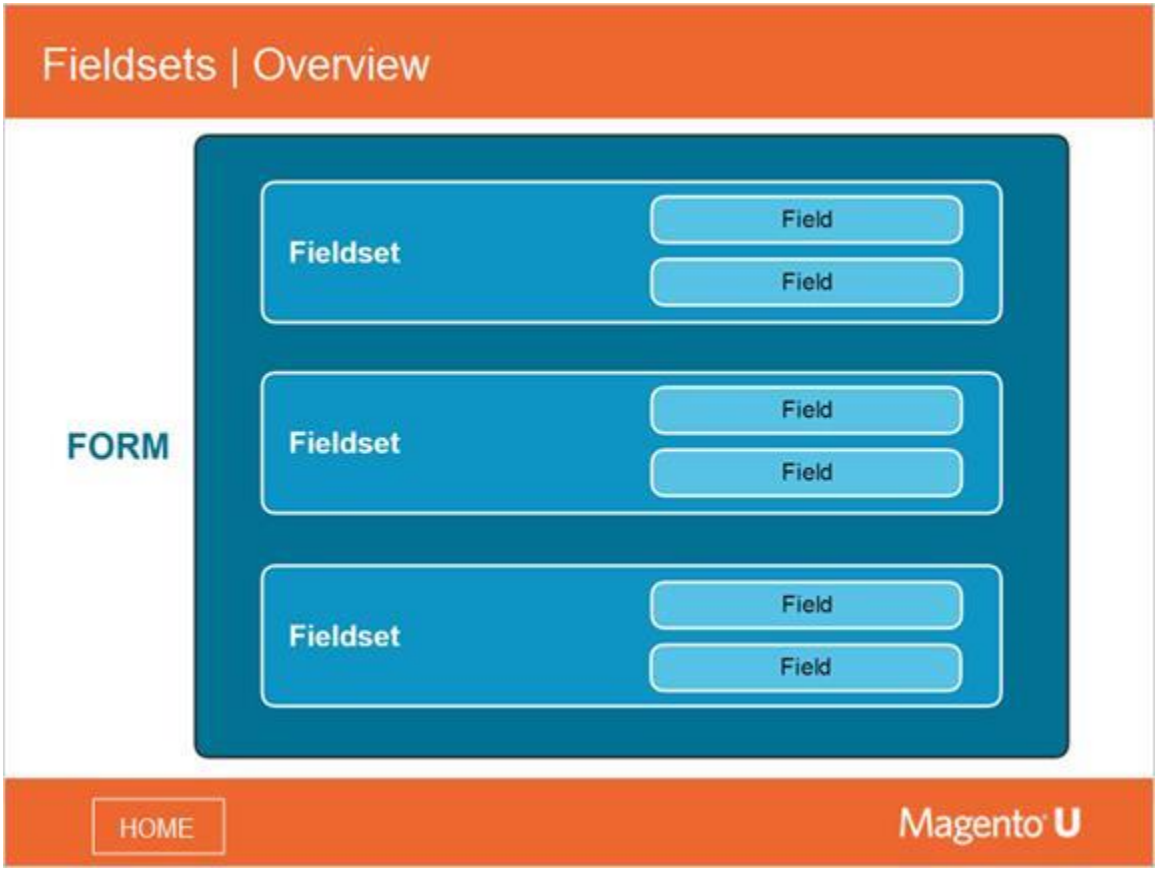


Notes:

Some key points about Magento 2 form's `dataProvider`.

5.16 Adminhtml Forms | Fieldsets

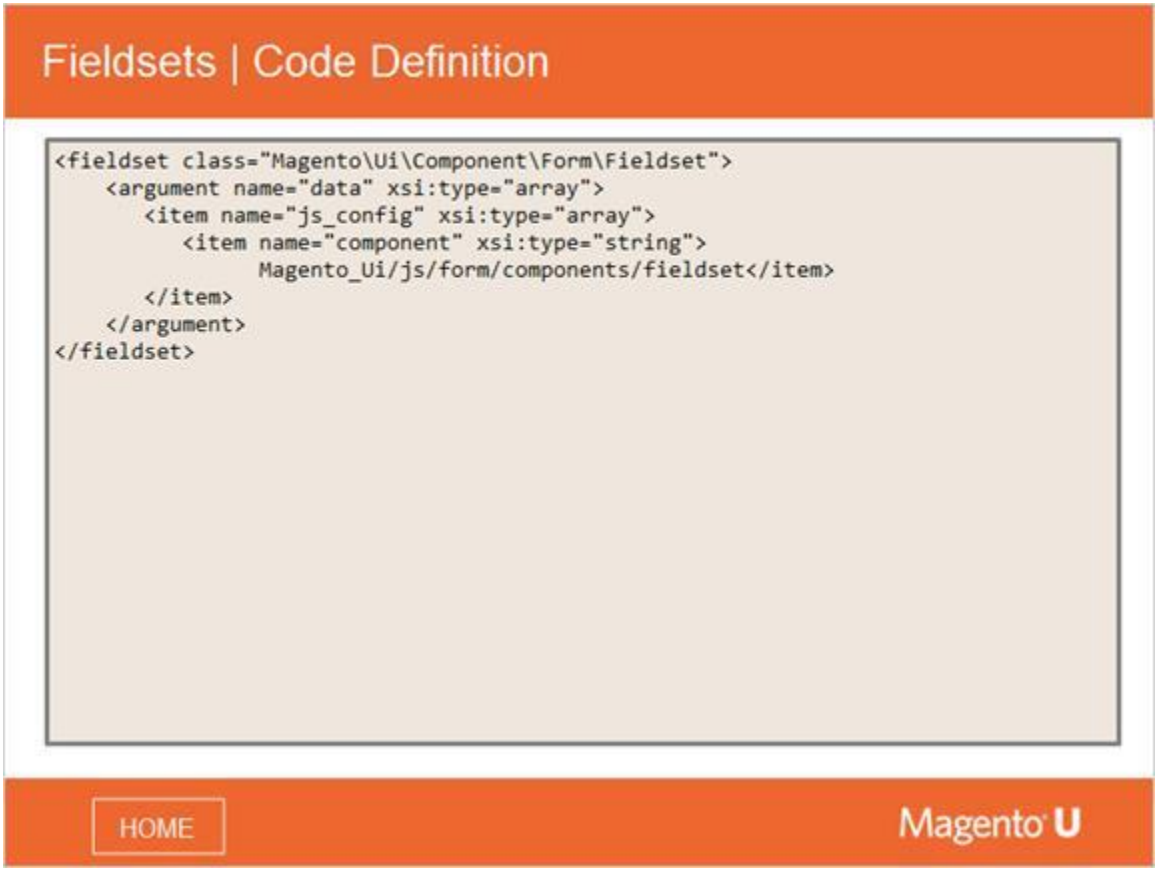
5.17 Fieldsets | Overview



Notes:

As a reminder, this diagram represents a typical form structure. Each form consists of one or more fieldsets, and each fieldset consists of a group of fields.

5.18 Fieldsets | Code Definition

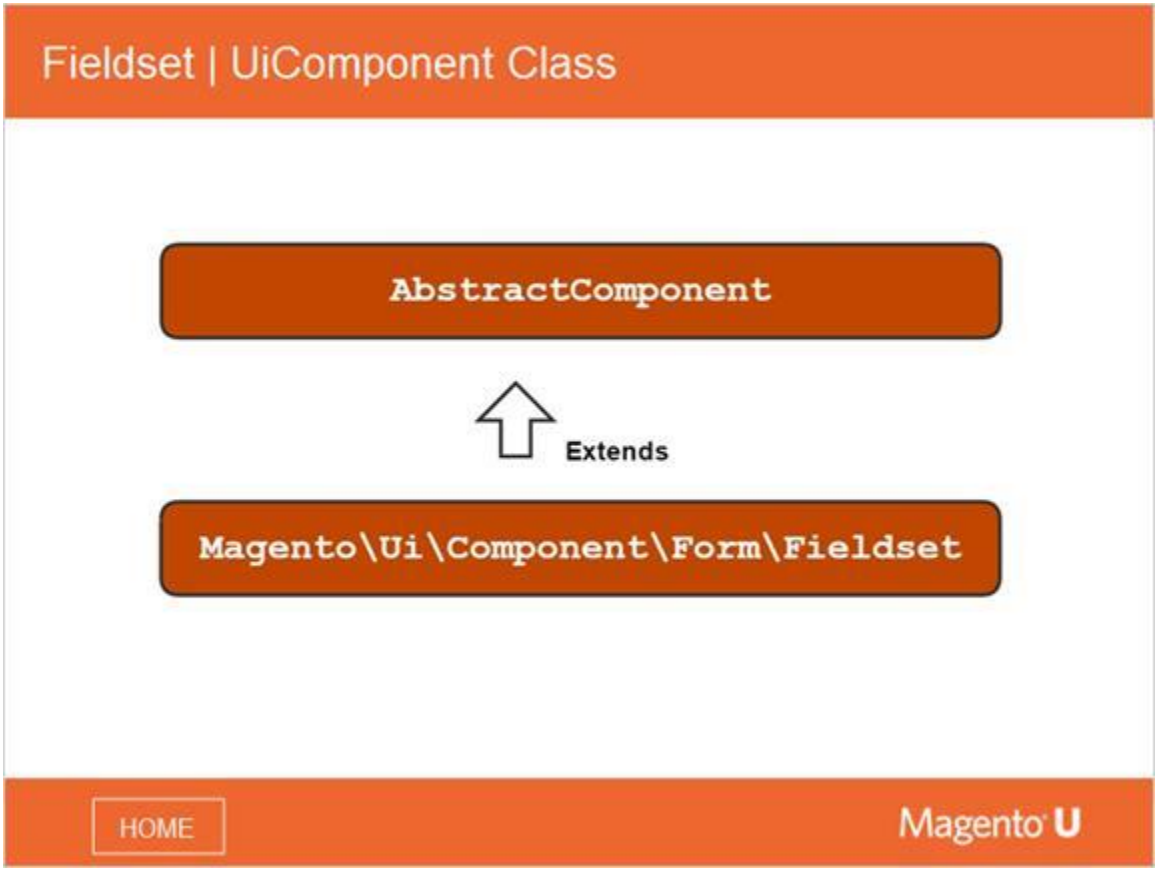


Notes:

Here is the original definition of a fieldset as a Ui component.

As you can see, it is a pretty standard Ui Component that is implemented in a `Magento/Ui/view/base/web/js/form/components/fieldset.js` JavaScript module.

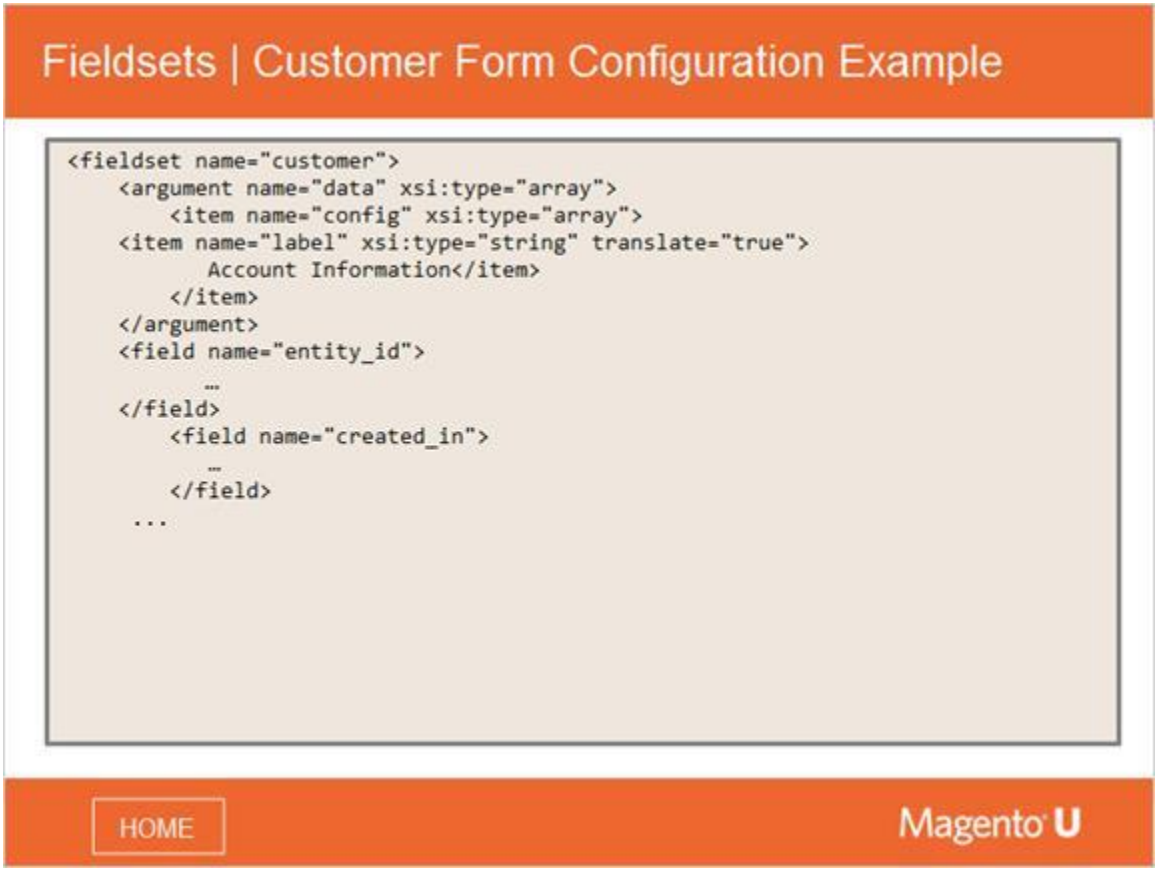
5.19 Fieldsets | UiComponent Class



Notes:

The fieldset component's PHP class extends `AbstractComponent`. This class is almost empty. Fieldset's configuration should be done in the form XML file, as we will see in the next slide.

5.20 Fieldsets | Customer Form Configuration Example

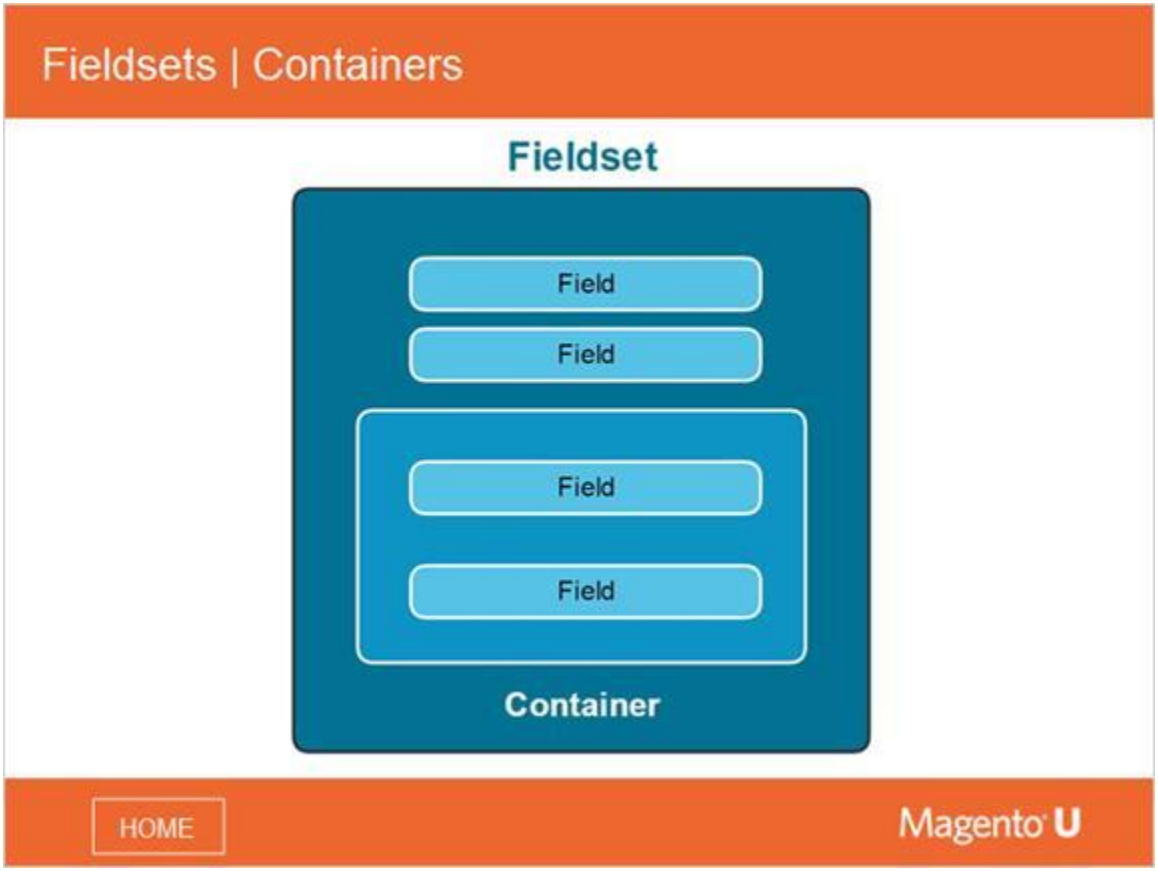


Notes:

The slide provides a customer form configuration example.

This form has many fieldsets, with each fieldset encapsulating field elements. As we see here, fieldset configuration consists of a data argument and a set of field nodes. Each node includes field configuration, which will be described in the slides that follow.

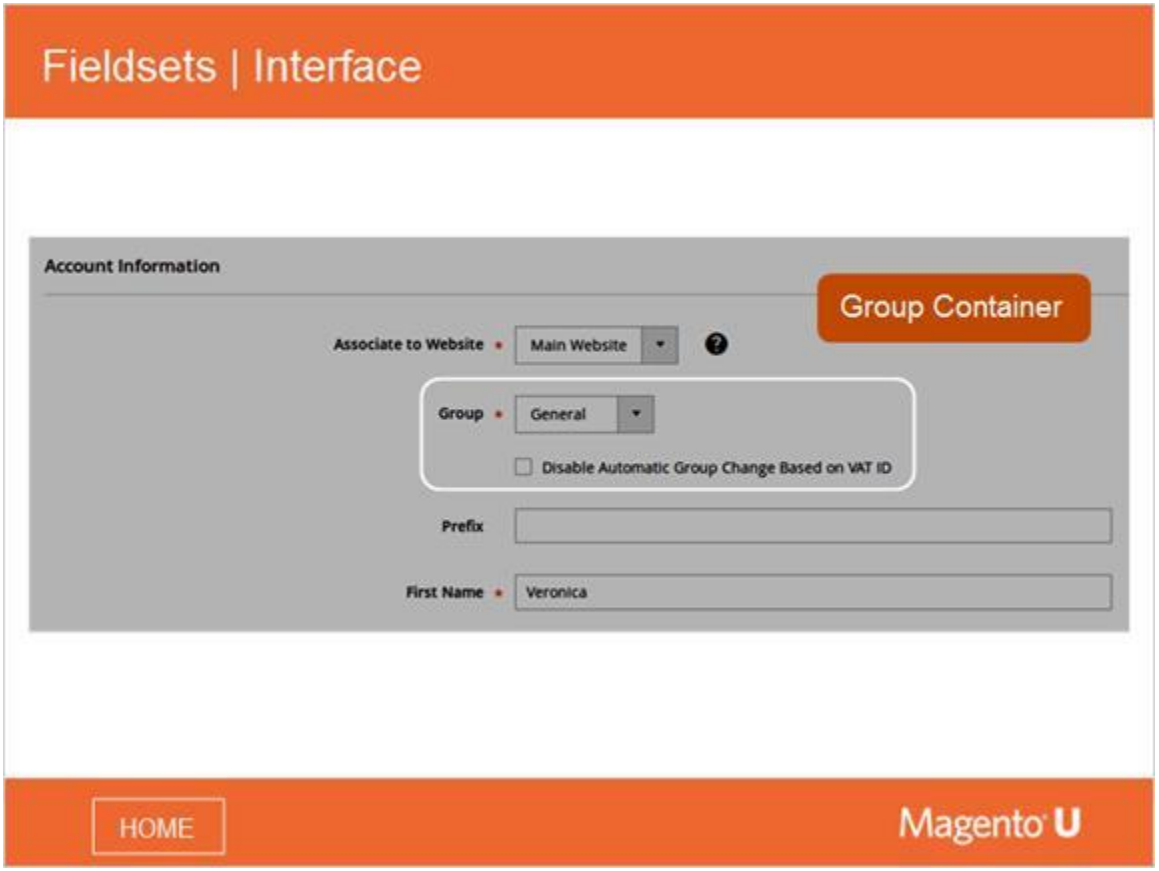
5.21 Fieldsets | Containers



Notes:

The role of a container is to group multiple fields inside of a fieldset. The next slide provides an example of when containers could be useful.

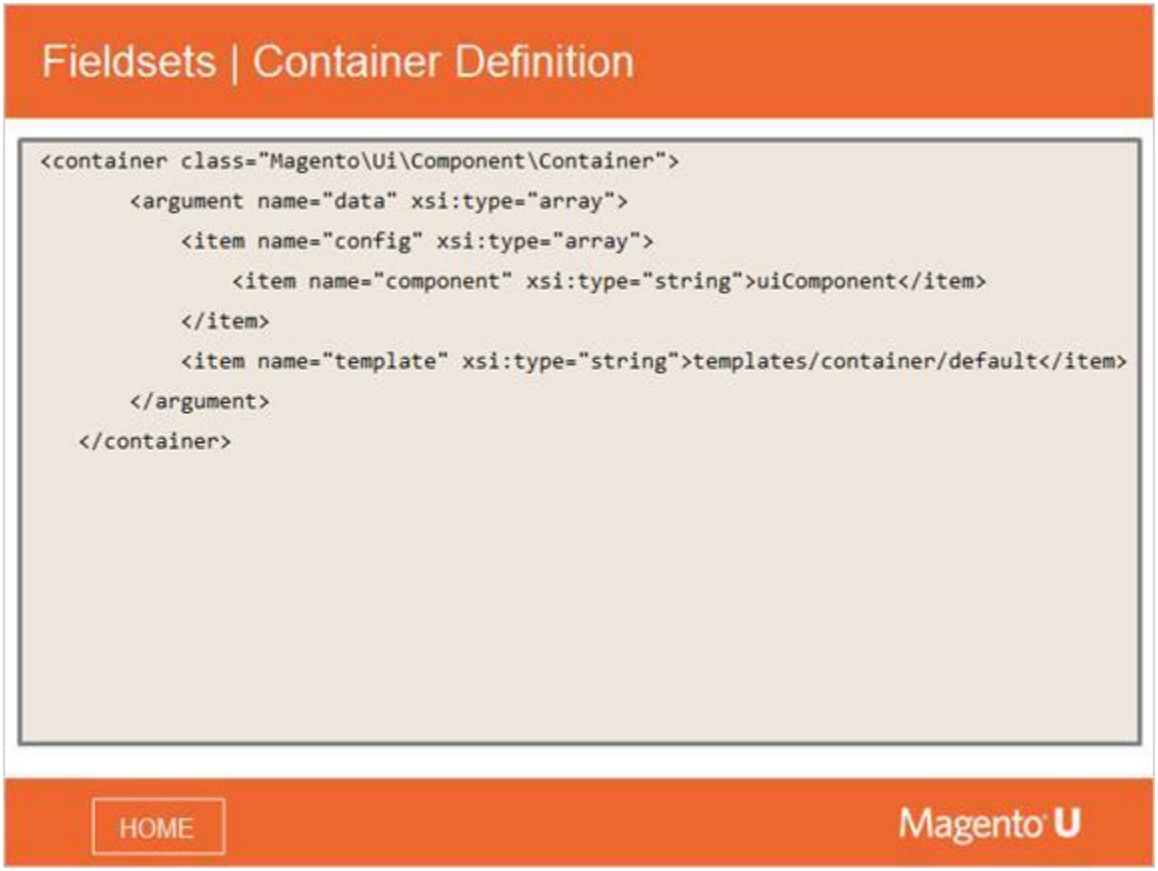
5.22 Fieldsets | Interface



Notes:

You can see two fields grouped here: the dropdown “General” and the checkbox “Disable Automatic Group Change Based on VAT ID.” Both are contained within the same label, “Group.”

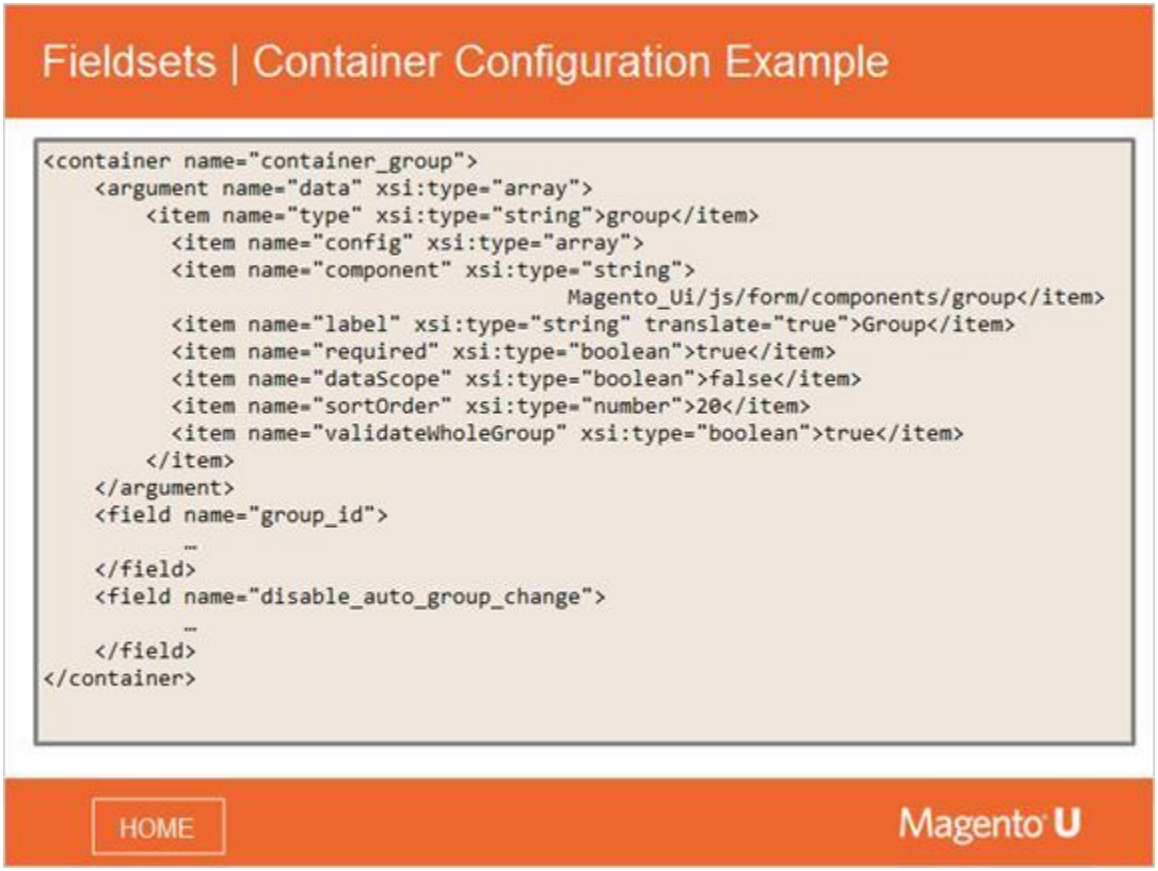
5.23 Fieldsets | Container Definition



Notes:

This code demonstrates a declaration of a Container UiComponent definition in `definition.xml`.

5.24 Fieldsets | Container Configuration Example



Notes:

This piece of XML code defines a group, the same as shown on the fieldsets interface diagram two slides earlier.

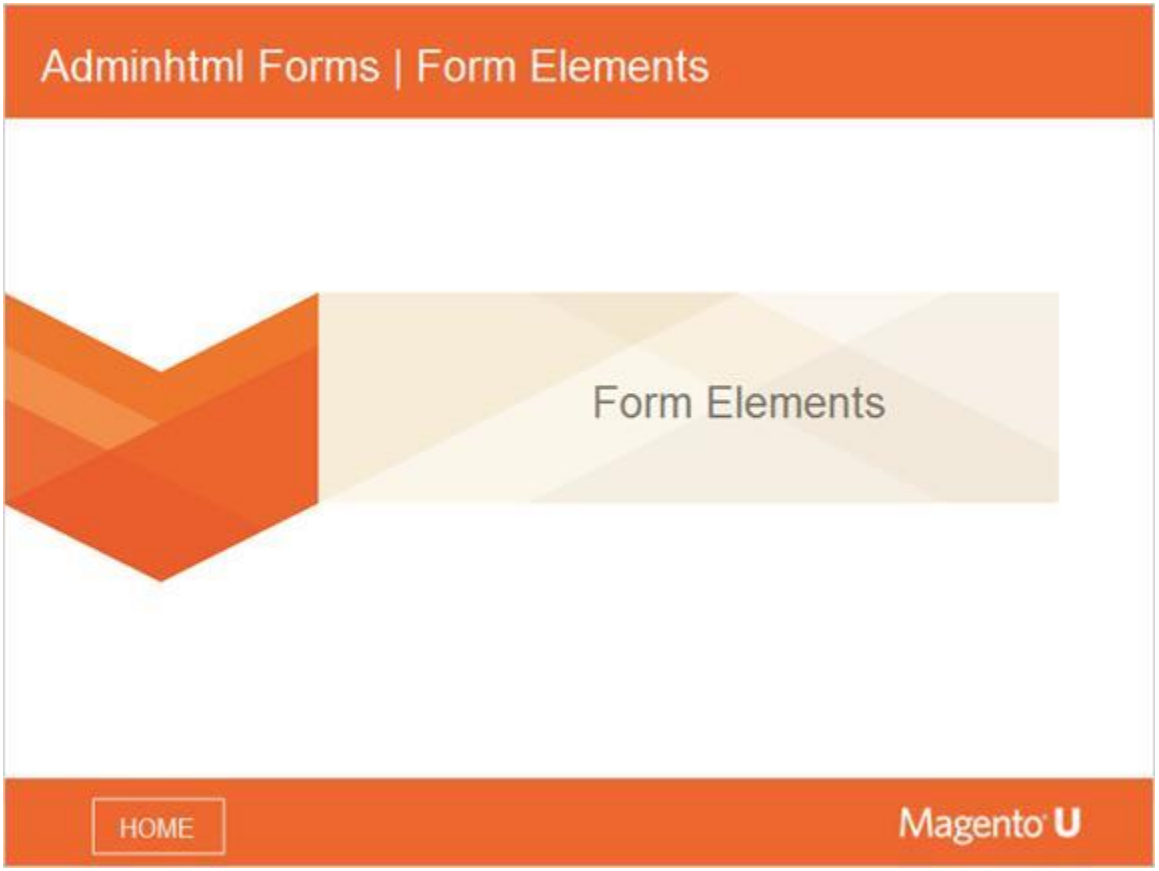
You can see it includes two fields and a set of parameters in the data argument.

5.25 Fieldsets | Container Class

Notes:

Here is an example of a Container PHP class. As you can see, it is relatively trivial.

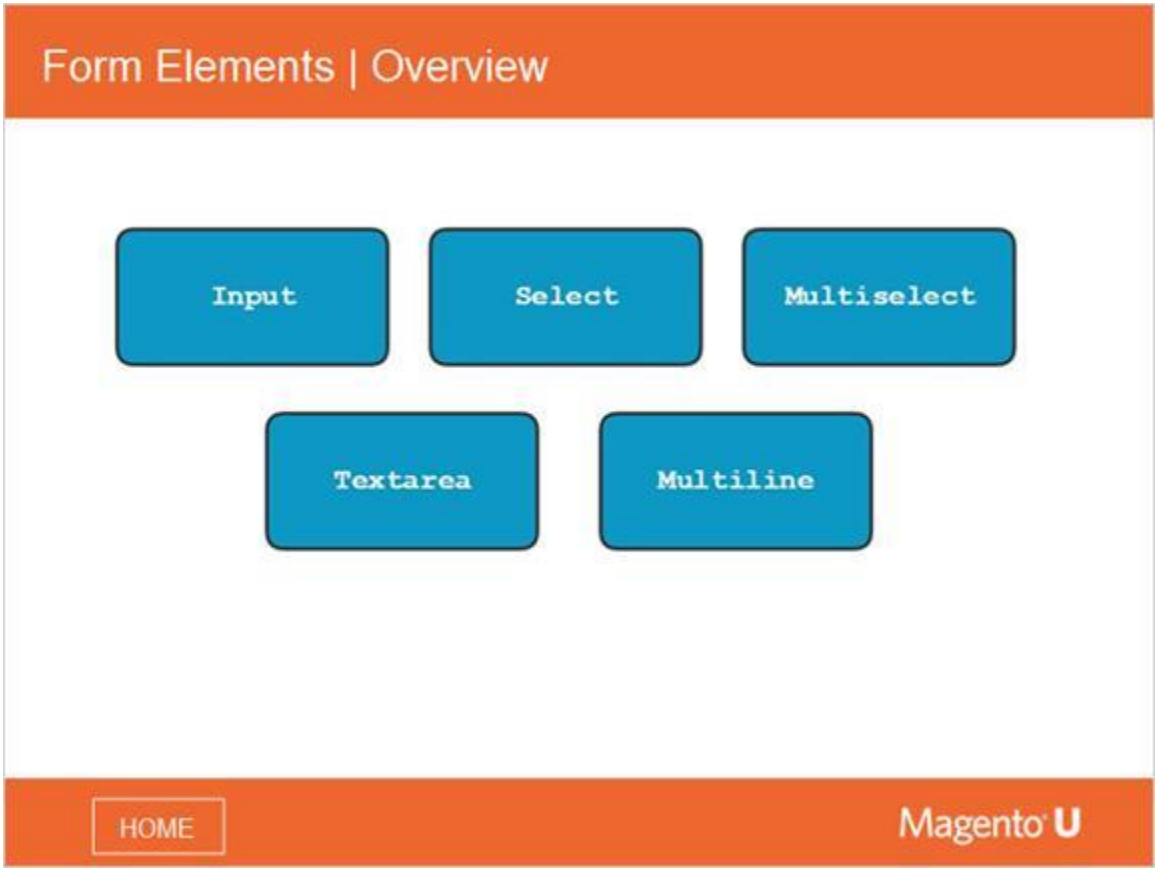
5.26 Adminhtml Forms | Form Elements



Notes:

And now, the final topic of this module: form elements.

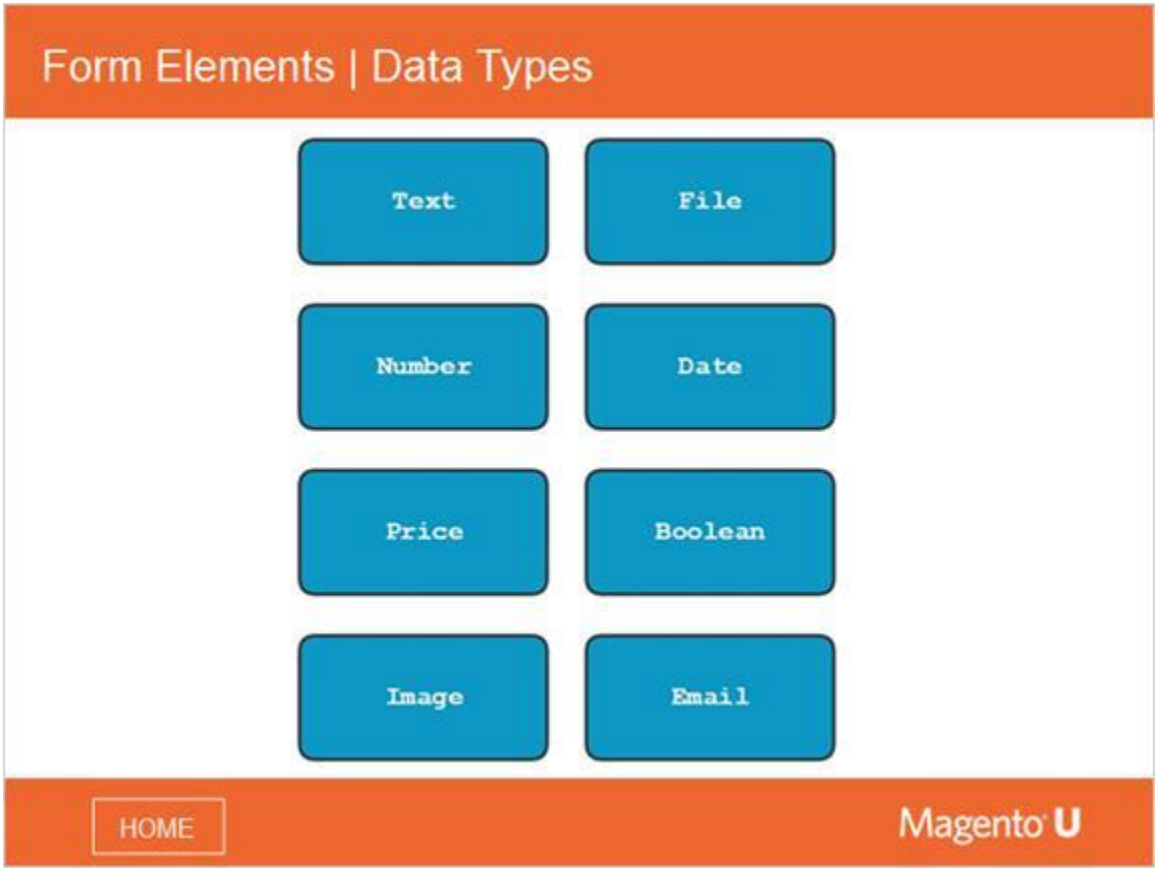
5.27 Form Elements | Overview



Notes:

This diagram presents the most important form elements. You can find the full list in the `definition.xml` file.

5.28 Form Elements | Data Types



Notes:

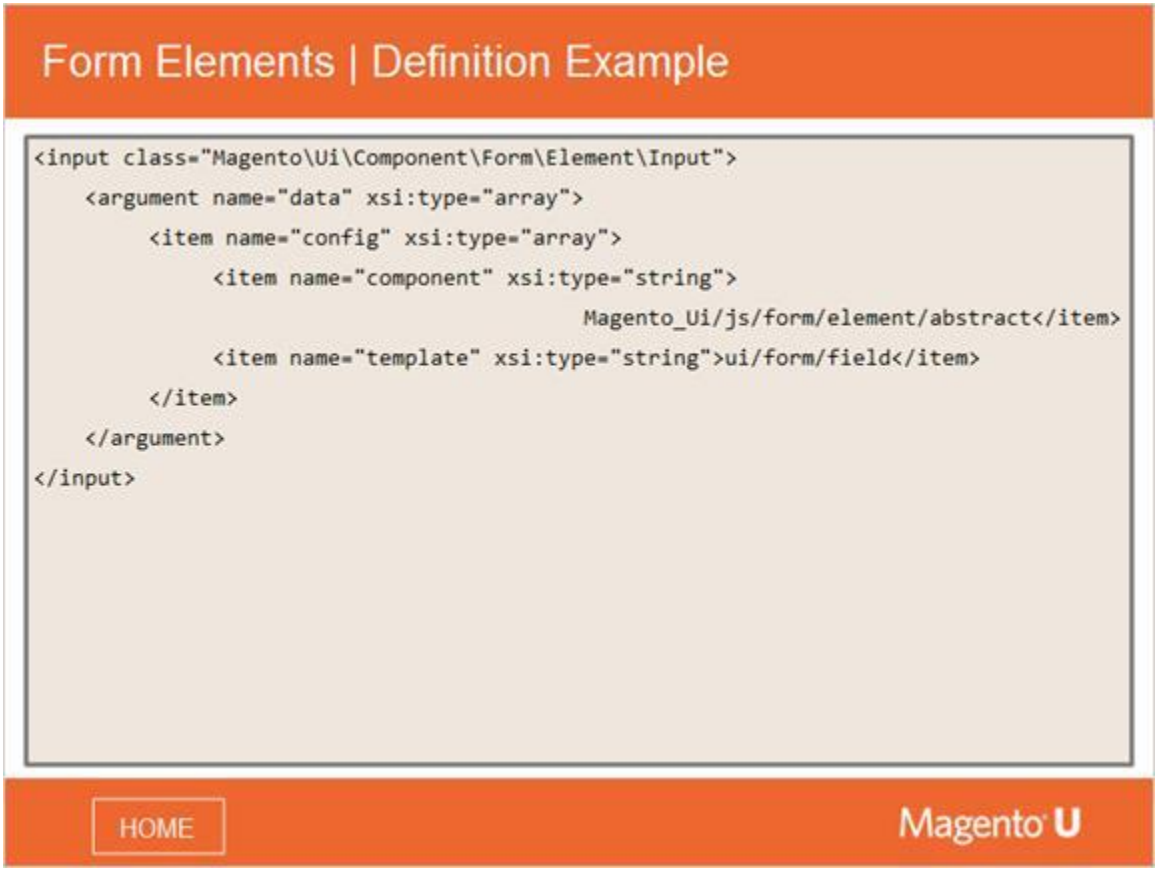
The slide shows a list of data types.

Note that data type does not explicitly define control, but rather how to treat data related to a certain field.

As an example, here is the code for an abstract data type class:

```
abstract class AbstractDataType extends AbstractComponent implements DataTypeInterface
{
    /**
     * Validate value
     *
     * @return bool
     */
    public function validate()
    {
        return true;
    }
}
```

5.29 Form Elements | Definition Example

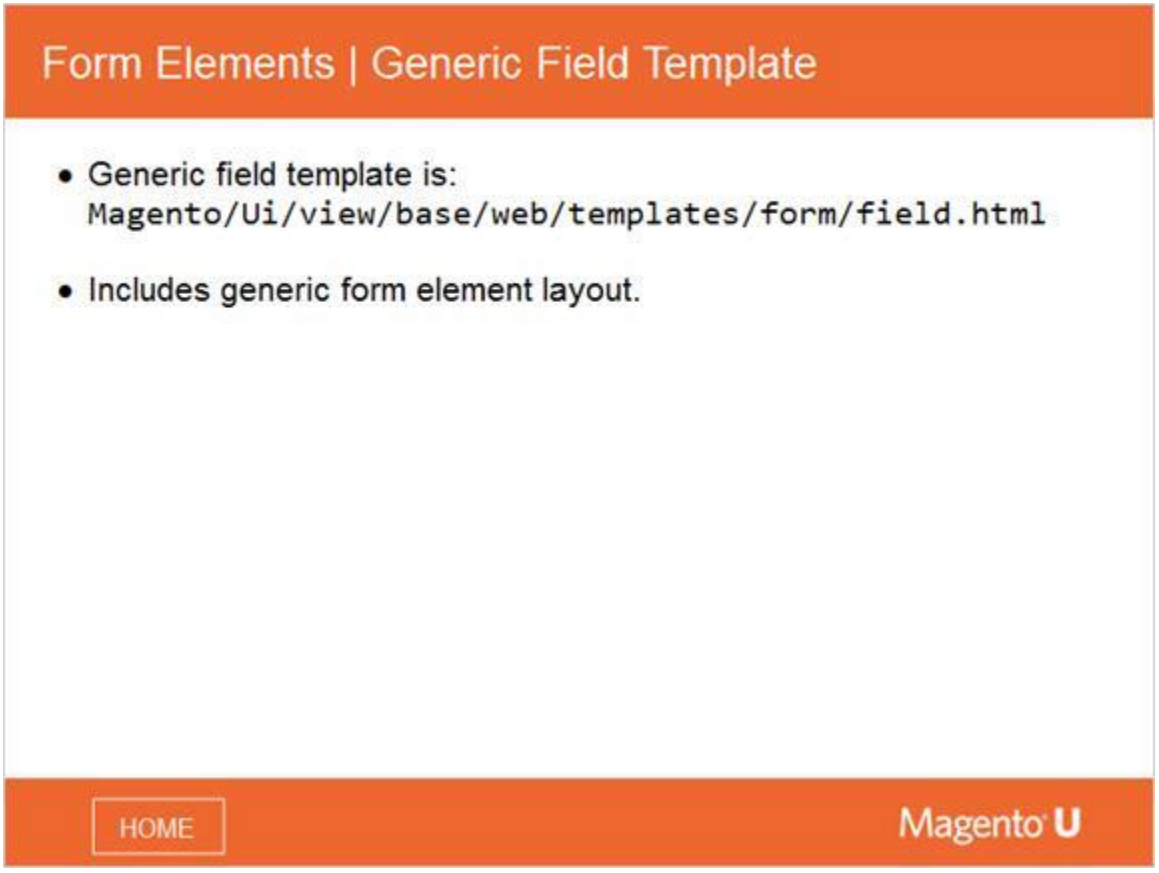


Notes:

Here we can see how form elements are defined in the `definition.xml` file.

There are three important elements to note:
- **Component PHP class:** `Magento\Ui\Component\Form\Element\Input`
- **JavaScript module:** `Magento/Ui/view/base/web/js/form/element/abstract.js`
- **Template:** `Magento/Ui/view/base/web/templates/form/element/input.html`

5.30 Form Elements | Generic Field Template



Notes:

A generic field template plays the same role as containers did in Magento 1. It wraps an element's template with auxiliary components like `before/after` triggers.

You can take a look at the corresponding code below:

```
<div class="admin__field"
     visible="visible"
     css="$data.additionalClasses"
     attr="'data-index': index">
    <label class="admin__field-label" if="$data.label" visible="$data.labelVisible" attr="for: uid">
        <span text="label" attr="'data-config-scope': $data.scopeLabel"/>
    </label>
    <div class="admin__field-control"
         css="'_with-tooltip': $data.tooltip, '_with-reset': $data.showFallbackReset &&
$data.isDifferedFromDefault">
        <render args="elementTmpl" ifnot="hasAddons()"/>

        <div class="admin__control-addon" if="hasAddons()">
            <render args="elementTmpl"/>

            <label class="admin__addon-prefix" if="$data.addbefore" attr="for: uid">
```

```
                <span text="addbefore"/>
            </label>
            <label class="admin__addon-suffix" if="$data.addafter" attr="for: uid">
                <span text="addafter"/>
            </label>
        </div>

        <render args="tooltipTpl" if="$data.tooltip"/>

        <render args="fallbackResetTpl" if="$data.showFallbackReset && $data.isDifferedFromDefault"/>

        <label class="admin__field-error" if="error" attr="for: uid" text="error"/>

        <div class="admin__field-note" if="$data.notice" attr="id: noticeId">
            <span text="notice"/>
        </div>

        <div class="admin__additional-info" if="$data.additionalInfo" html="$data.additionalInfo"></div>

        <render args="$data.service.template" if="$data.hasService()"/>
    </div>
</div>
```

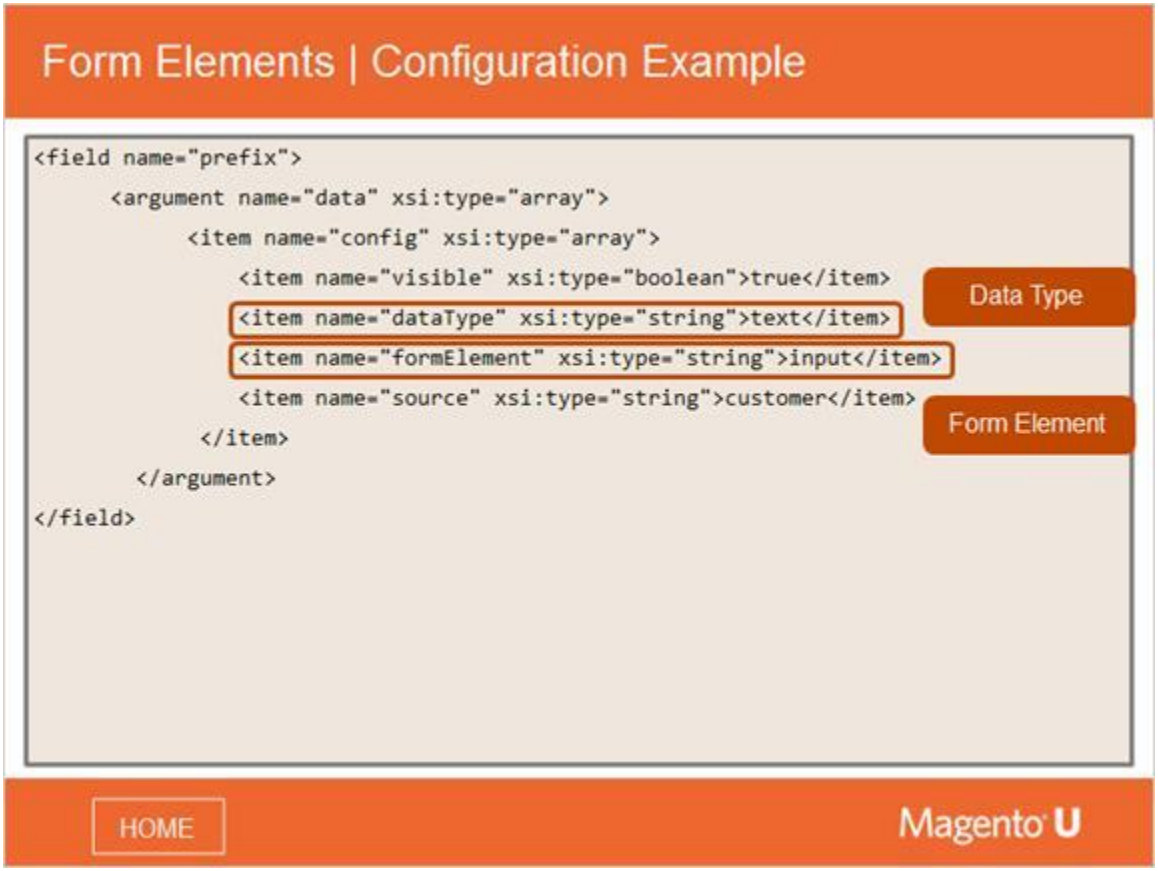
5.31 Form Elements | Element Template

Notes:

This code presents an example of an element's template using input.html, which was included in the `definition.xml` slide shown earlier.

As you can see, it is a knockout template composed of certain data (`value, name, id`) that are inserted using the generic UiComponent engine.

5.32 Form Elements | Configuration Example



Notes:

Finally, we can take a look at the practical application of the theory above.

This slide shows an example of real input form field declaration for the customer module.

Note the two major configuration options that define the template's look and feel:
- `dataType`
- `formElement`

5.33 Form Elements | Select Example

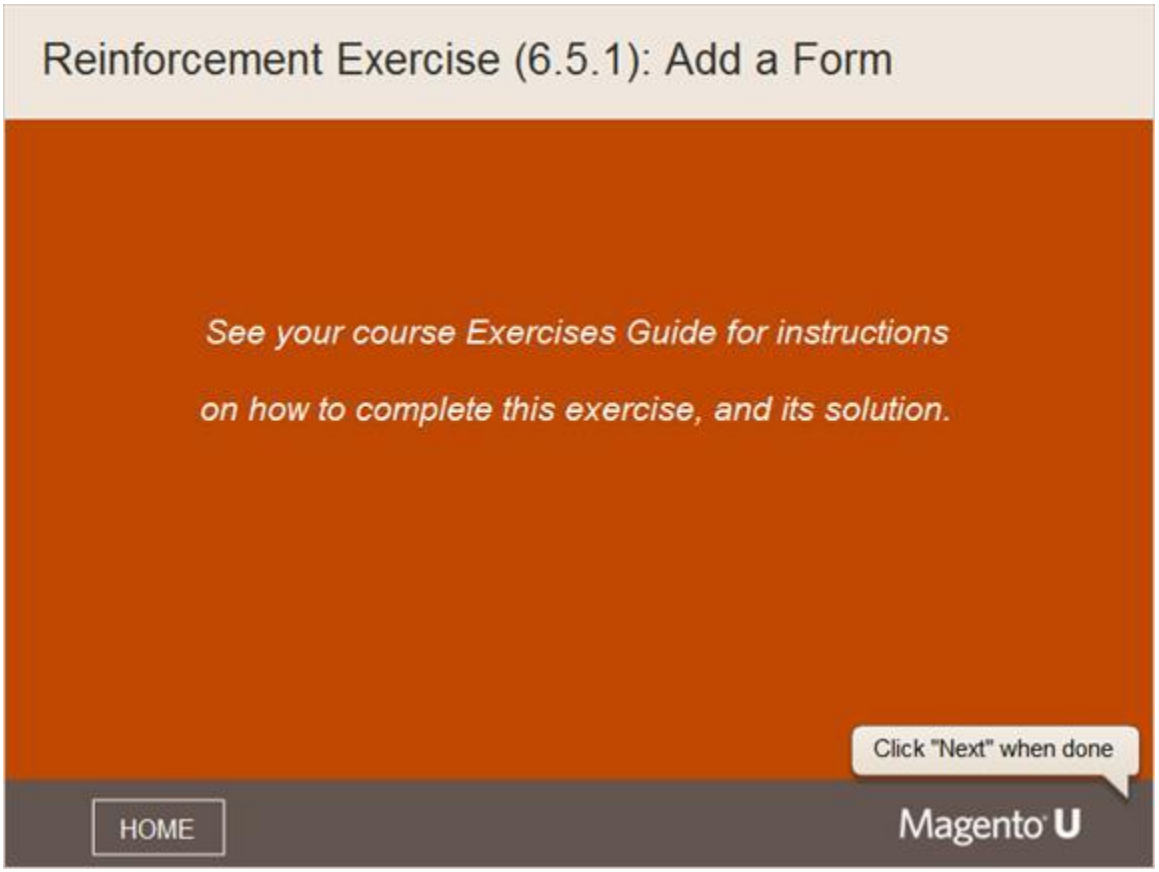
Notes:

Another example, this time for the `select` field.

You can see more configuration options in this example, such as `validation` (which defines the CSS class used for validation) and `tooltip` (which shows a message on mouse-over).

Note that `dataType` for select-type fields defines the type of select values.

5.34 Reinforcement Exercise (6.5.1)

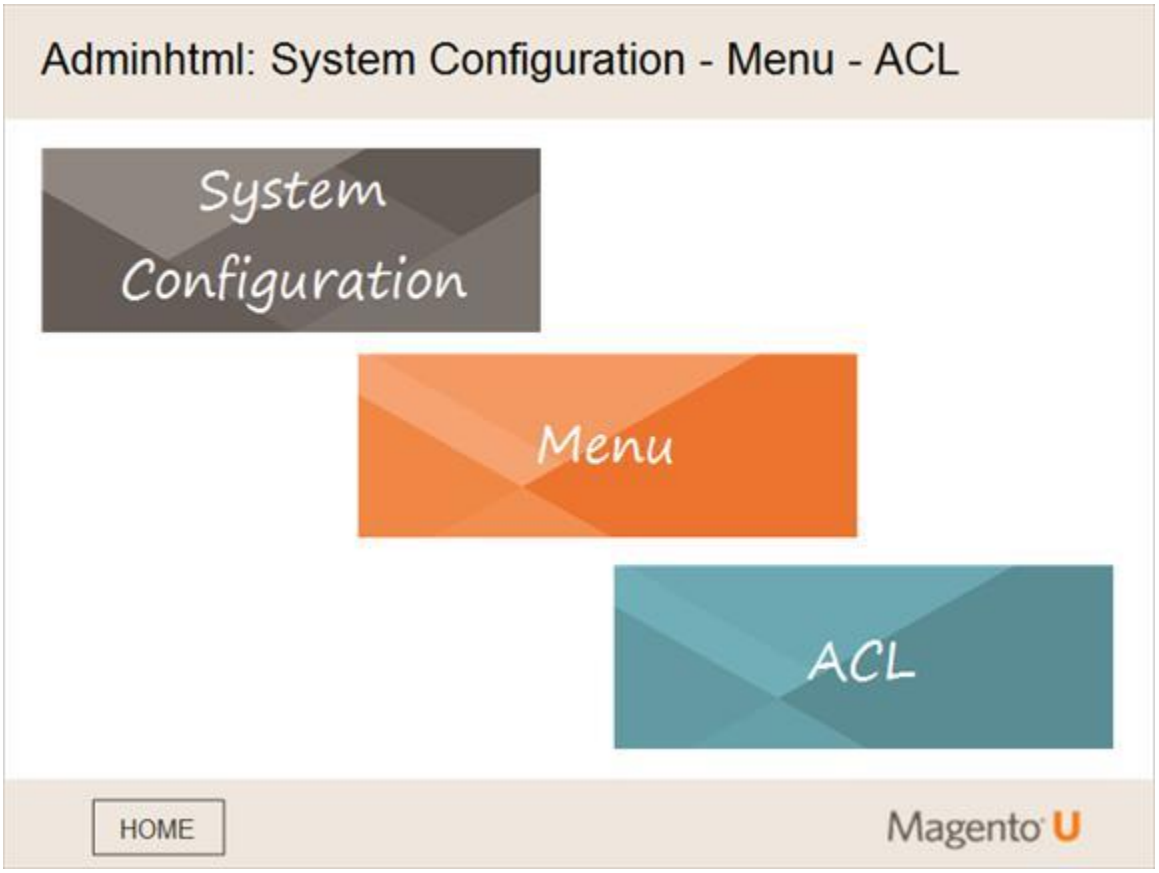


Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

6. Configuration

6.1 Adminhtml: System Configuration - Menu - ACL



Notes:

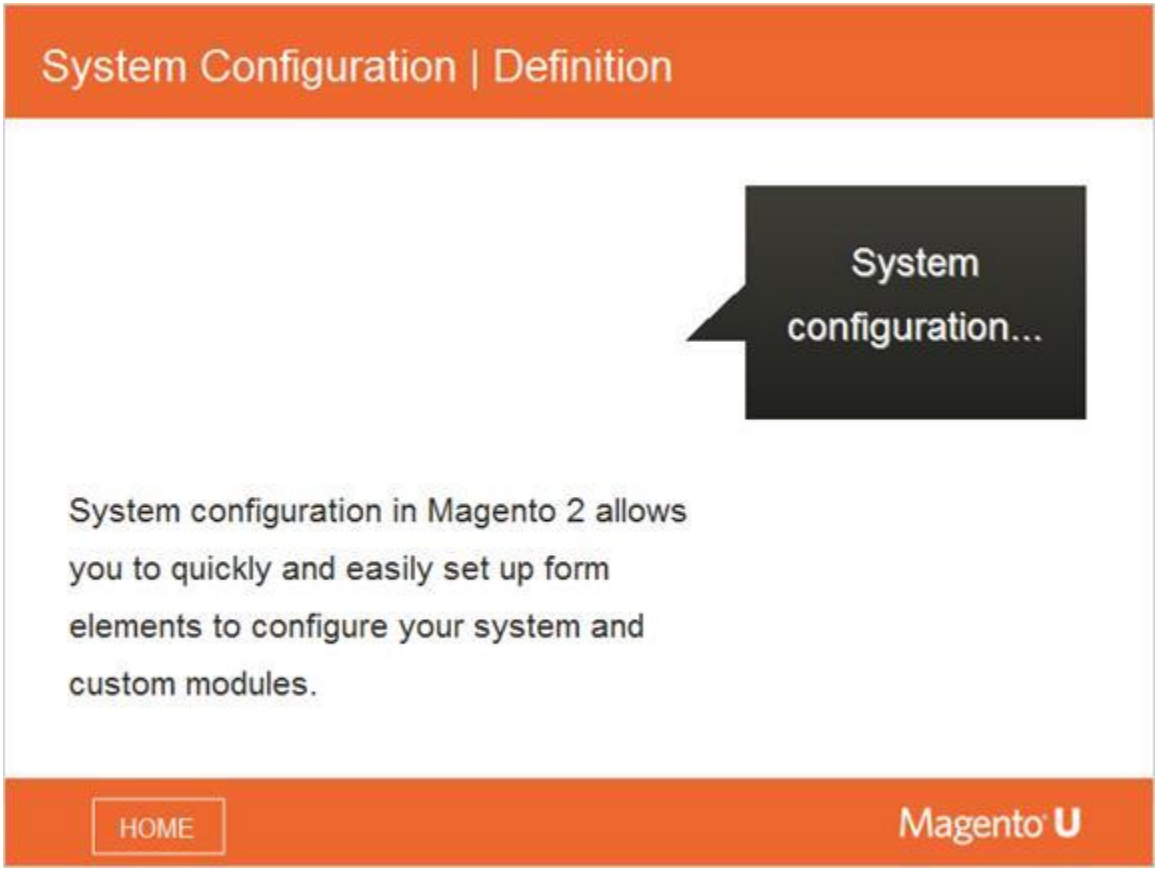
Now that we have finished our general discussion of the framework API, we are going to look at the API in more detail, focused on repositories and the business logic API.

6.2 Module Topics | System Configuration - Menu - ACL

Notes:

In this module, we will discuss in detail...
- System configuration
- Menus
- ACL permissions

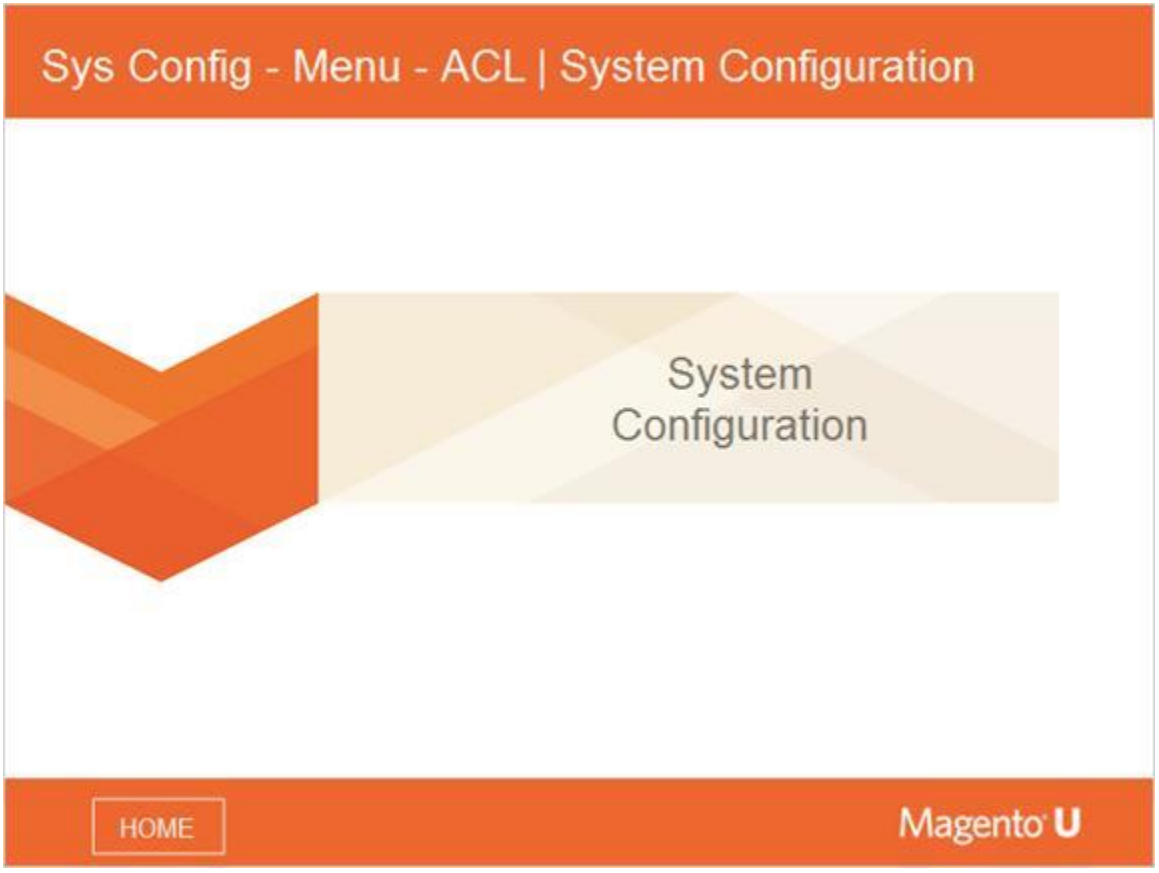
6.3 System Configuration | Definition



Notes:

System configuration in Magento 2 allows you to quickly and easily set up form elements to configure your system and custom modules.

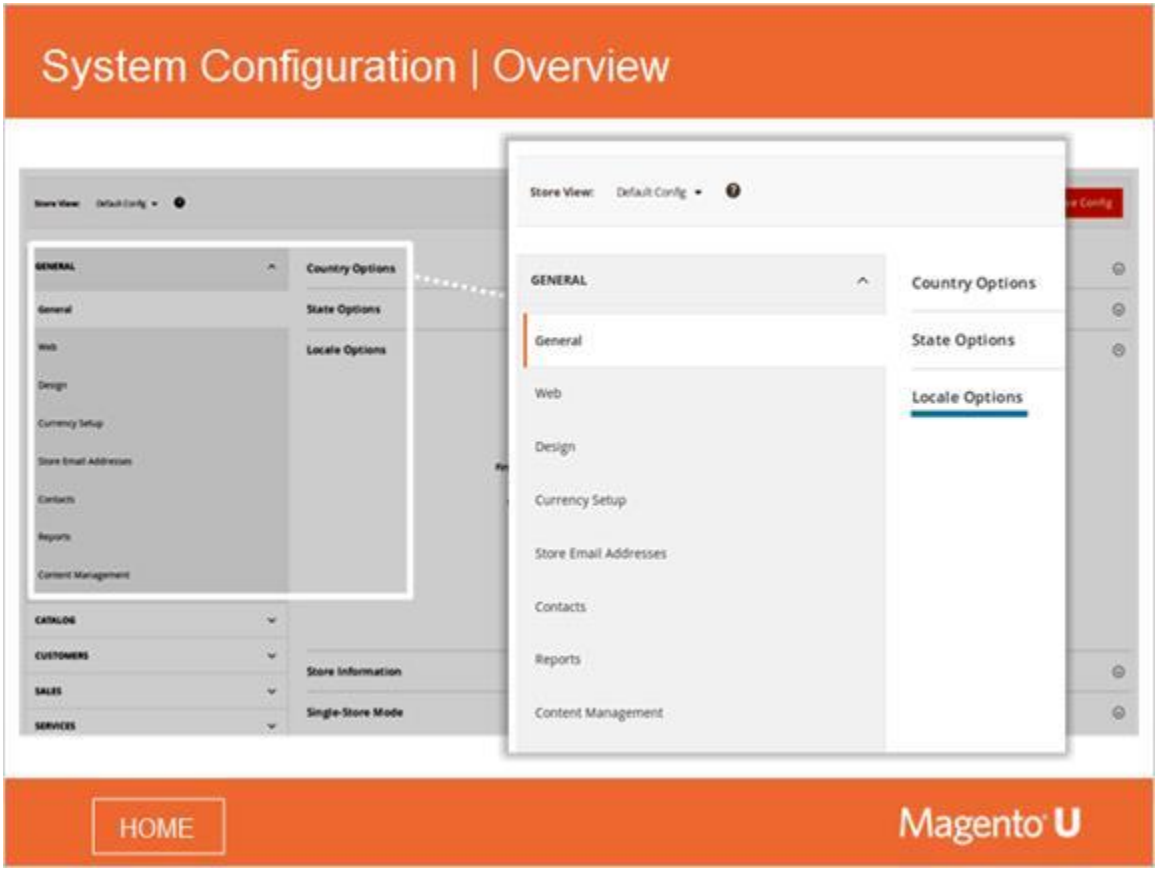
6.4 Sys Config - Menu - ACL | System Configuration



Notes:

We'll begin with a discussion of system configuration within Magento 2.0.

6.5 System Configuration | Overview



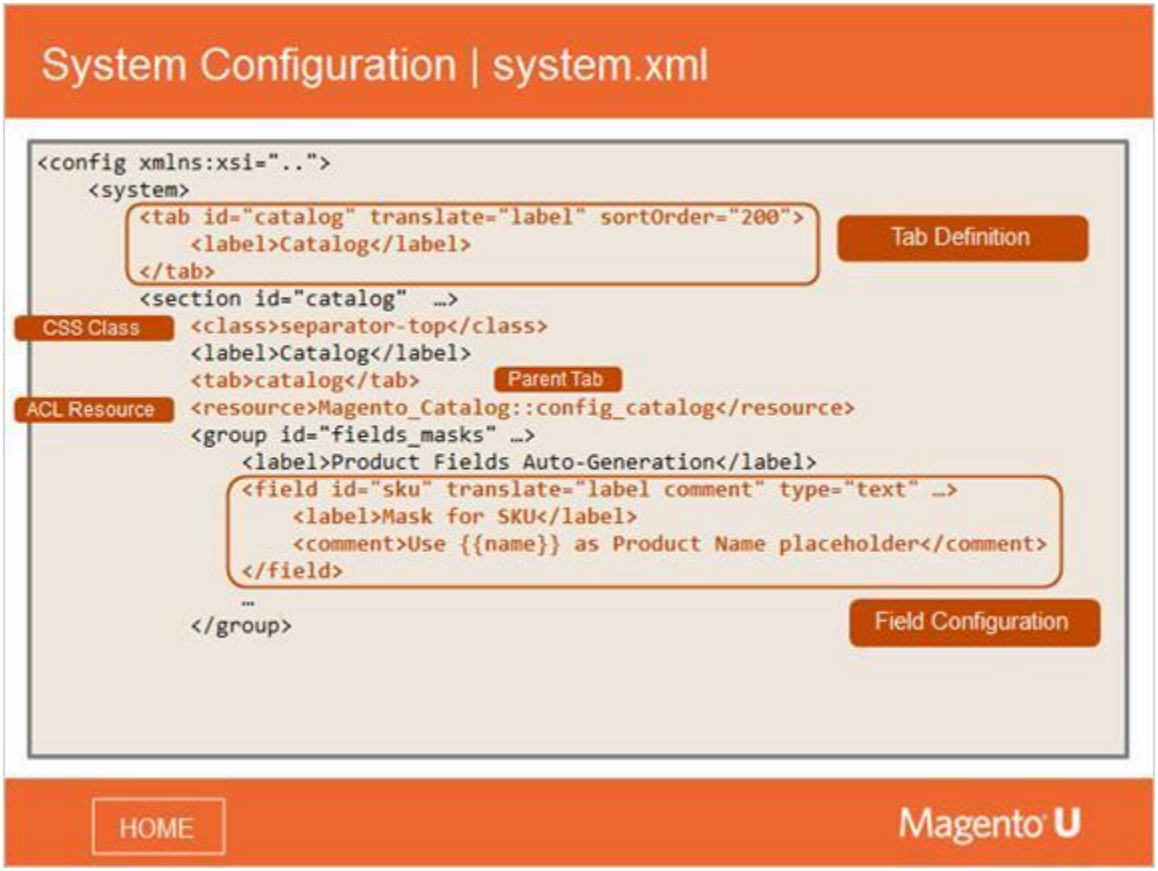
Notes:

The system configuration page in Magento 2 provides a list of available options.

Here are some important things to note:

- Configuration options are for the merchant, so technical aspects that the merchant may not understand are not included.
  For example, the order of **calculating** totals for a shopping cart is technical information and is not presented in system configuration, while the order of **displaying** totals is included.
- Every configuration option value from this page is stored in a database (unlike XML configuration). The corresponding table is `core_config_data`, as in Magento 1.
- Configuration options can be scoped, on global, website, and store levels; scope changing is possible by selecting a scope in the top left dropdown of the page.
- All configuration options are previously declared in a special configuration file: `system.xml`.
- Option names consist of 3 parts: section, group, field ID.
  For example, the “Locale” option presented on this slide will have the name: `general/locale/code`.

6.6 System Configuration | system.xml



Notes:

On this page, we can see a typical configuration for the system configuration. Let's explore its structure.

First of all, configuration is located in `etc/adminhtml/system.xml`. Its XSD schema is located in `Magento/Config/etc/system_file.xsd`

Two types of second-level nodes are available in the system.xml file: tab and section.

In the tab node, you can define a new tab for the configuration. The example on the slide demonstrates the “Catalog” tab definition.

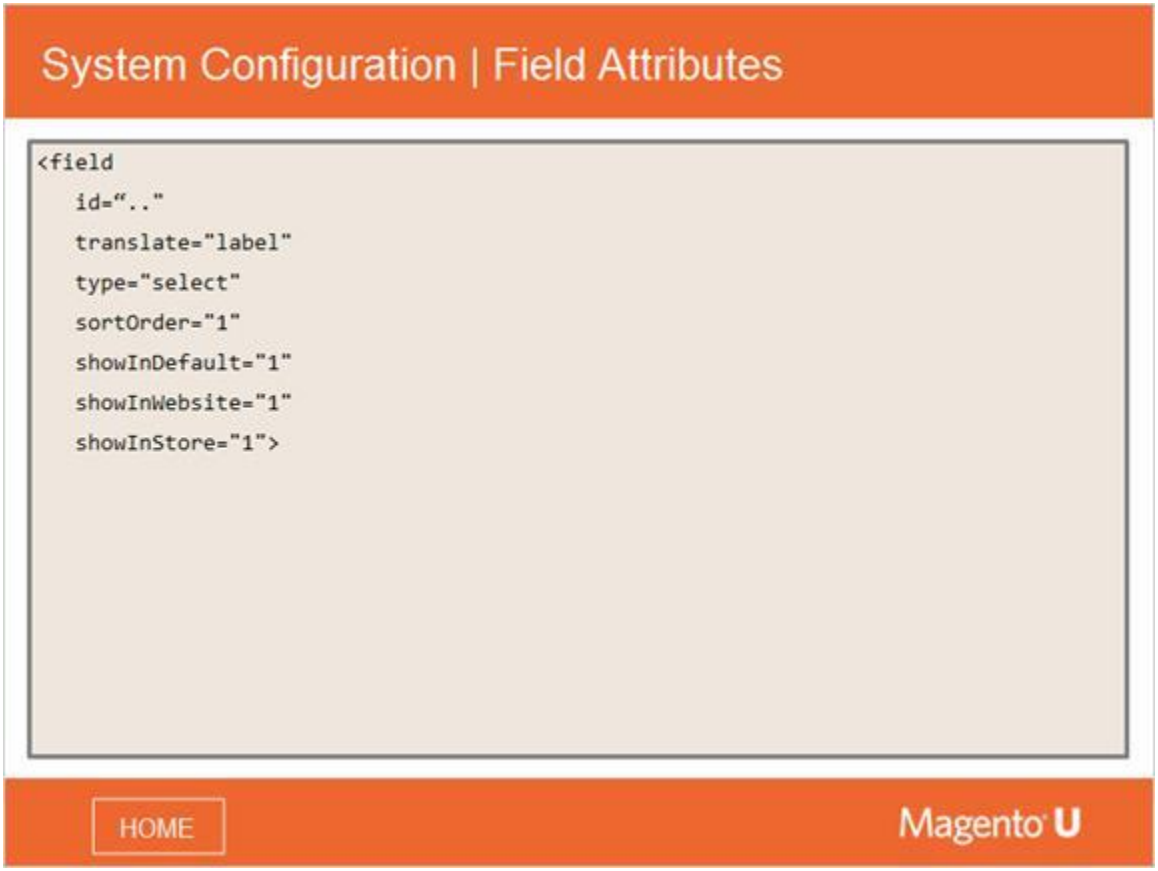
The section node is where most of the options are described. It defines (1) which tab the section belongs to (in this case, catalog), and (2) the resource (ACL resource and group). The section node also has a name (second-level after Tab).

Group provides third-level option groupings (after Tab). It is only displayed on the right side of a page, not the left side (called an "accordion"). Each group belongs to the section it is declared in (as does the tab), has its own label, and has a list of fields.

Field is actually an option in itself. It might have different scope (defined by the field node attributes, discussed later). Each field has its name (`id attribute`), label (`<label>` child node), and type (`type` attribute).

You can also optionally include other child subnodes like `source_model` and `frontend_model`. They will be covered later in the course as well.

6.7 System Configuration | Field Attributes



Notes:

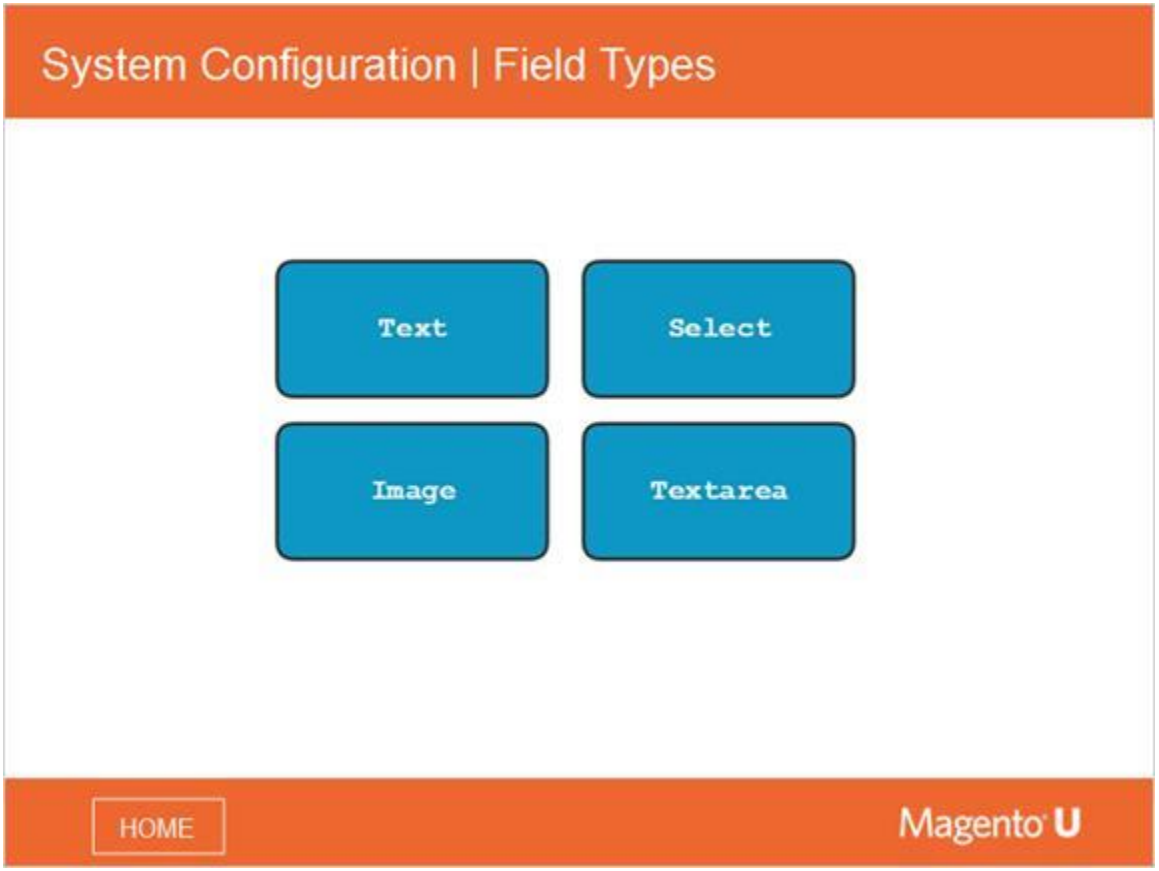
Let's go deeper into the structure of `system.xml`. On the slide, you can see the following field node attributes:

- **`Id`** - A field name. It is a third part of an option's name (after section and group).
- **`Translate`** - A standard translation attribute. As in Magento 1, it defines which child node to translate.
- **`Type`** - Defines what kind of control has to be used to render an option. On the next slide you will see the available types.
- **`SortOrder`** - The order inside an accordion.
- **`showIn*`** - Defines a scope.

Example:
If `showInDefault=1`, `showInWebsite=0`, `showInStore=0`, the option will be global and will not be visible on the website/store level.

If `showInDefault=1`, `showInWebsite=1`, `showInStore=0`, the option will be visible on default and website scopes, but its scope will be website (which means you can define a default value but then you can redefine the value on the website level).

6.8 System Configuration | Field Types



Notes:

Here is a list of standard field types. For non-standard fields, you can use the `frontend_model` child node, which is described later.

6.9 System Configuration | Field Children

Notes:

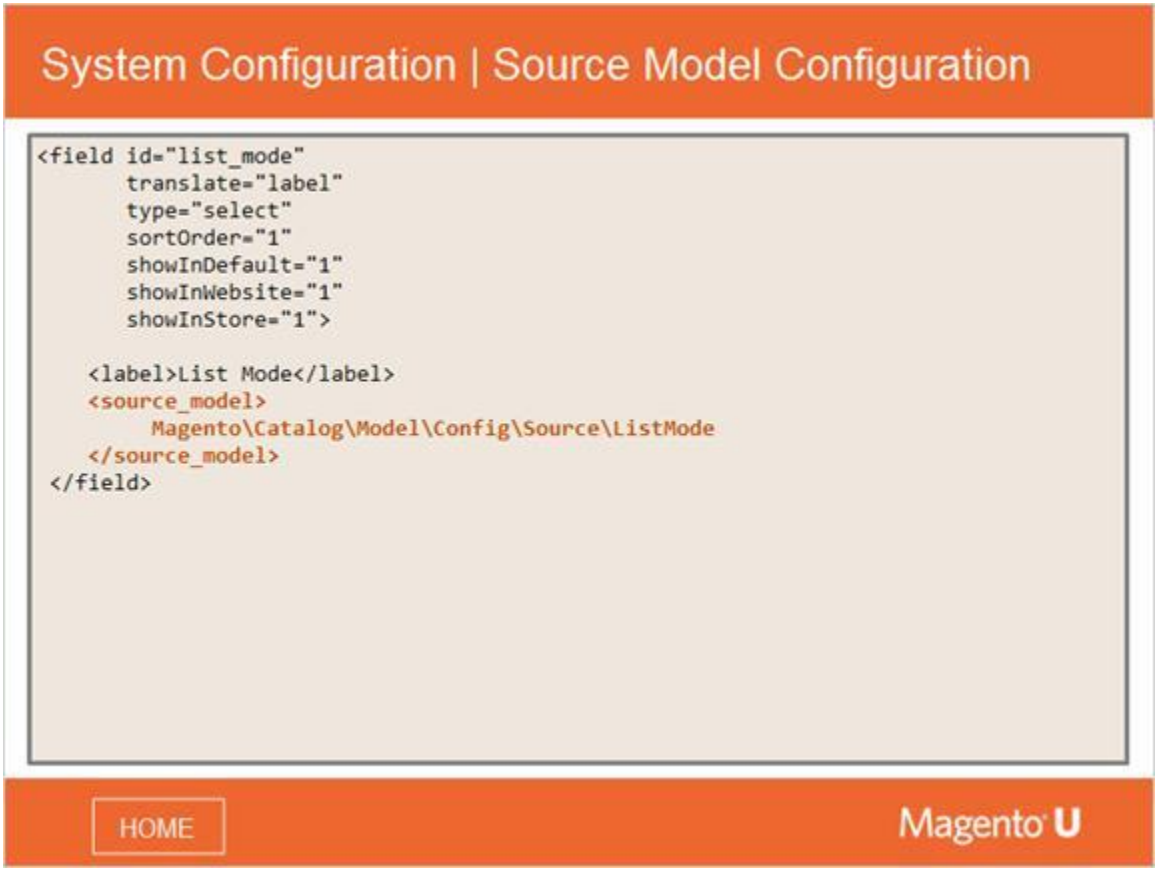
The slide shows a list of all the available `<field>` node children. The list is taken from an `xsd` file: `Magento/Config/etc/system_file.xsd`.

Note the following:

- **`label:`** The visible name of an option.
- **`comment:`** Text written in a small font below an option input. For example: `Catalog/Storefront/Products per Page on Grid Allowed Values;` a comment is "`comma-separated`".
- **`tooltip:`** Draws a small question mark, which shows a message on mouse hover. For example, see `General/Design/Design Theme/User-agent` exceptions; it is defined in `Magento/Backend/etc/adminhtml/system.xml`.
- **`hint:`** Not used in the native Magento.
- **`frontend_class:`** CSS class to assign to the option. For example: `General/Store Information/Country`, defined in `Magento/Backend/etc/adminhtml/system.xml.`
- **`frontend_model:`** Used to define custom content for an option; will be discussed in more detail later in the course.
- **`backend_model:`** A class which allows an action to be performed when an option is saved. For example: class `Magento\Config\Model\Config\Backend\Email\Address`.
- **`source_model:`** Used to provide a list of values for select-type configuration values.
- **`config_path:`** Defines an alternative path to the option, instead of the standard three-levels path. This alternate path is used to save an option in a database; for example: `Magento/Paypal/etc/adminhtml/system.xml, merchant_country` field.
- **`validate:`** CSS class used for validation. For example: `General/Store Email Addresses/Custom Email 1/Sender Email,` defined in `Magento/Backend module.`

- **`can_be_empty:`** Flag which defines whether an option can have no value.
- **`if_module_enabled:`** Inherited from Magento 1, a condition that defines an option only when particular modules are enabled.
- **`base_url:`** Used for image-type fields.
- **`button_url, more_url:`** Specific properties; in the native Magento, used in the PayPal module.
- **`demo_url:`** Not used in the native Magento.
- **`button_label:`** A specific property; used only with a custom frontend model (for example, in the PayPal module).
- **`hide_in_single_store_mode:`** A property that defines whether a field should be available for a single store mode. For example, `price_scope` doesn't make sense in such a mode.
- **`upload_dir:`** Used for image-type fields.
- **`source_service:`** Not currently used in Magento.
- **`depends:`** Defines other nodes on which this node depends. For example, see `Magento/Paypal/etc/adminhtml/system/express_checkout.xml`.
- **`attribute:`** Allows the definition of a custom field's attribute. For example, see `Magento/Paypal/etc/adminhtml/system/express_checkout.xml`, and `Magento/Paypal/Block/Adminhtml/System/Config/ApiWizard.php`.
- **`requires:`** Makes a field available only if another field/group is activated. For example, look at the PayPal module `Magento/Paypal/etc/adminhtml/system/express_checkout.xml.`

6.10 System Configuration | Source Model Configuration

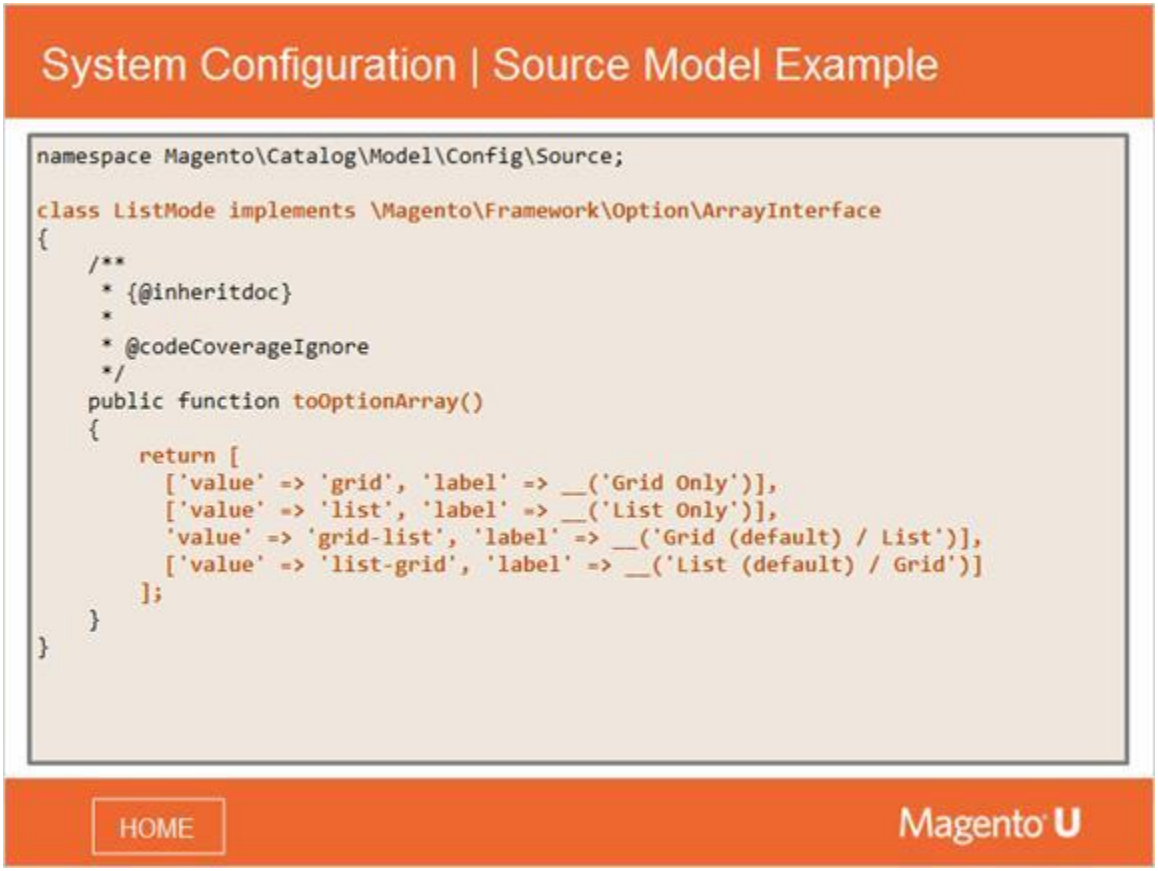


Notes:

Here is a code example of a source model configuration.

As mentioned earlier, the source model is used to provide a list of available values for dropdown-type configuration options. On the next slide is an example of such a model implementation.

6.11 System Configuration | Source Model Example



Notes:

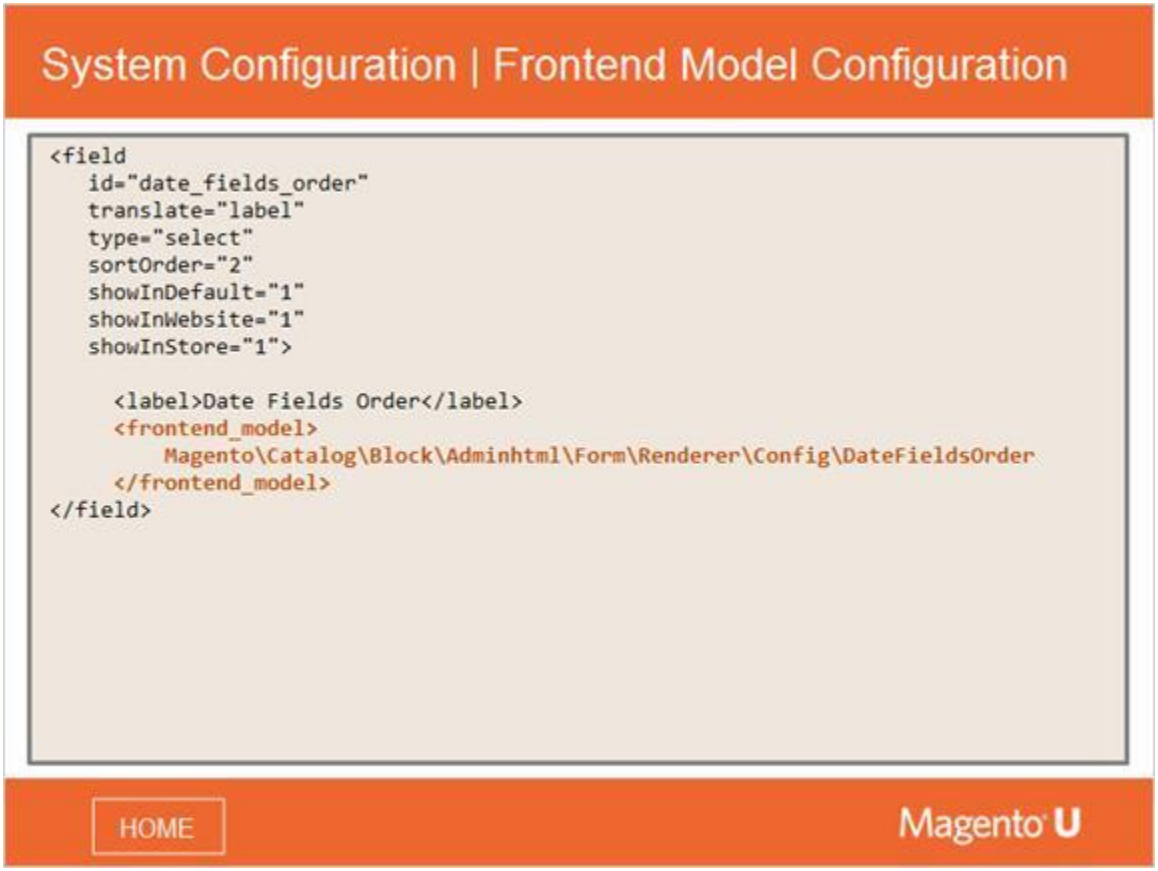
Here we can see an example of a source model.

The main goal of these classes is to provide a list of available options for select-type configuration fields.

A class has to implement `\Magento\Framework\Option\ArrayInterface`, which requires the implementation of the `toOptionArray()` method.

This method should return an array of values and labels in the format shown. Note that the "__" (double underscore) is now a global function that provides translation.

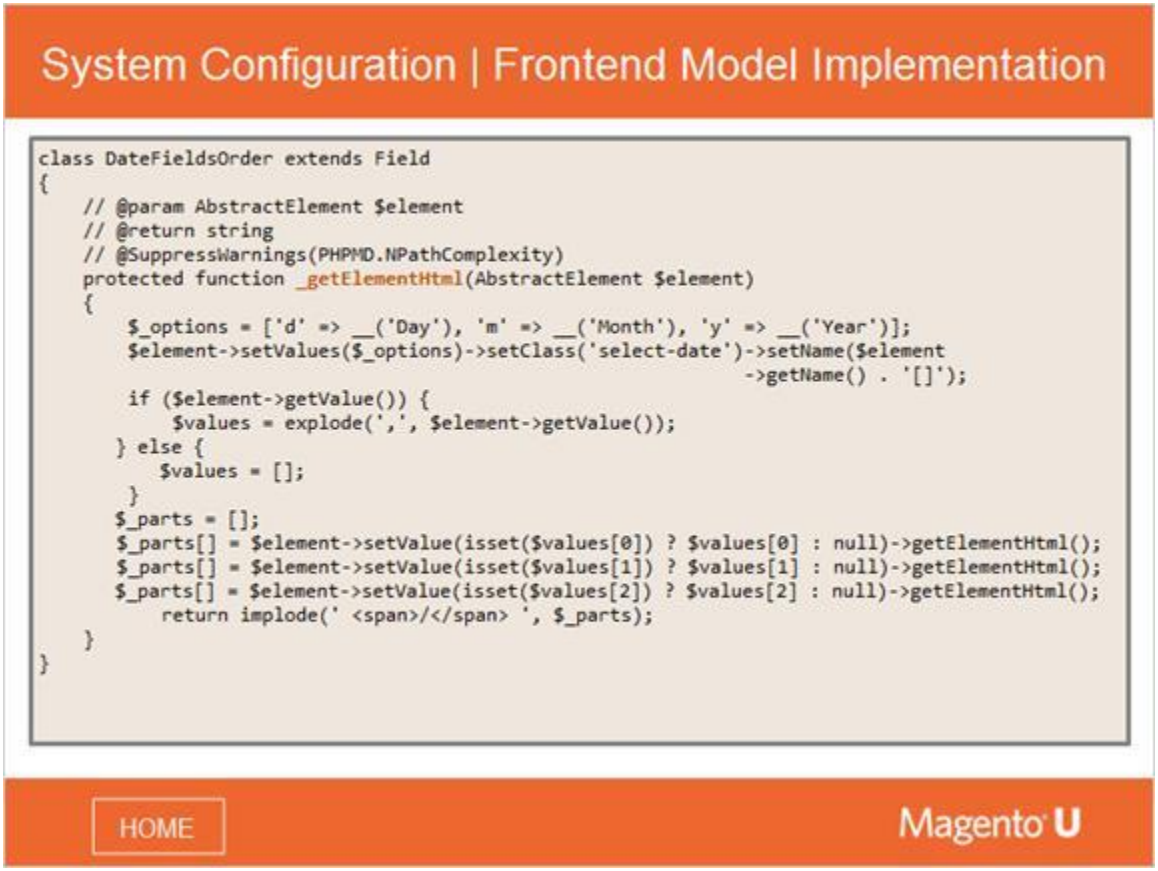
6.12 System Configuration | Frontend Model Configuration



Notes:

The code provides an example of a frontend model configuration. A frontend model is used to render a custom element for a field.

6.13 System Configuration | Frontend Model Implementation



Notes:

Here is an example of a frontend model implementation. You can see that it extends `Magento\Config\Block\System\Config\Form\Field`, which itself extends `\Magento\Backend\Block\Template`. Note that the overridden `_getElementHtml()` method generates the "value" part of an option (control).

You can better understand how the frontend model works by looking at the `render()` method (defined in the `Magento\Config\Block\System\Config\Form\Field` class). The highlighted code shows which methods can be overridden in the frontend model, and what effect that would have.

```
    public function render(\Magento\Framework\Data\Form\Element\AbstractElement $element)
        {
            $isCheckboxRequired = $this->_isInheritCheckboxRequired($element);

            // Disable element if value is inherited from other scope. Flag has to be set before the value is
    rendered.
            if ($element->getInherit() == 1 && $isCheckboxRequired) {
                $element->setDisabled(true);
            }

            $html = '<td class="label"><label for="' .
                $element->getHtmlId() .
                '">' .
```

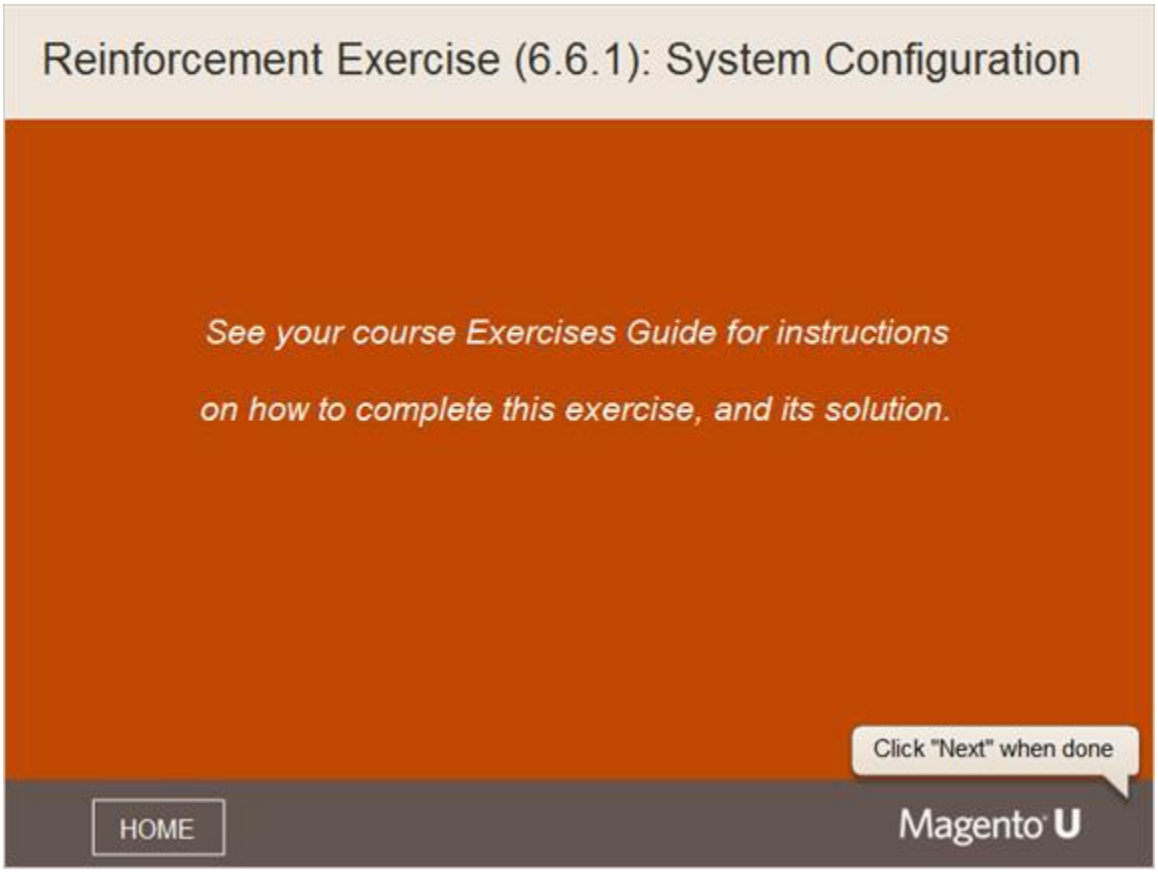
```
                $element->getLabel() .
                '</label></td>';
            $html .= $this->_renderValue($element);

            if ($isCheckboxRequired) {
                $html .= $this->_renderInheritCheckbox($element);
            }

            $html .= $this->_renderScopeLabel($element);
            $html .= $this->_renderHint($element);

            return $this->_decorateRowHtml($element, $html);
        }

     protected function _renderValue(\Magento\Framework\Data\Form\Element\AbstractElement $element)
        {
            if ($element->getTooltip()) {
                $html = '<td class="value with-tooltip">';
                $html .= $this->_getElementHtml($element);
                $html .= '<div class="tooltip"><span class="help"><span></span></span>';
                $html .= '<div class="tooltip-content">' . $element->getTooltip() . '</div></div>';
            } else {
                $html = '<td class="value">';
                $html .= $this->_getElementHtml($element);
            }
            if ($element->getComment()) {
                $html .= '<p class="note"><span>' . $element->getComment() . '</span></p>';
            }
            $html .= '</td>';
            return $html;
        }
    ...
```

The `$element` object (an instance of the `AbstractElement` class) is an instance of a field configured in `system.xml`, so we have access to its properties. For example, in the next example (from the Magento/Paypal module), you can see the use of field properties from `system.xml`:

```
namespace Magento\Paypal\Block\Adminhtml\System\Config;

class BmlApiWizard extends ApiWizard
{
    /**
     * Path to block template
     */
    const WIZARD_TEMPLATE = 'system/config/bml_api_wizard.phtml';

    /**
     * Get the button and scripts contents
     *
     * @param \Magento\Framework\Data\Form\Element\AbstractElement $element
     * @return string
     */
    protected function _getElementHtml(\Magento\Framework\Data\Form\Element\AbstractElement $element)
    {
        $originalData = $element->getOriginalData();
        $this->addData(
            [
                'button_label' => __($originalData['button_label']),
```

```
                'button_url' => $originalData['button_url'],
                'html_id' => $element->getHtmlId(),
            ]
        );
        return $this->_toHtml();
    }
}
```

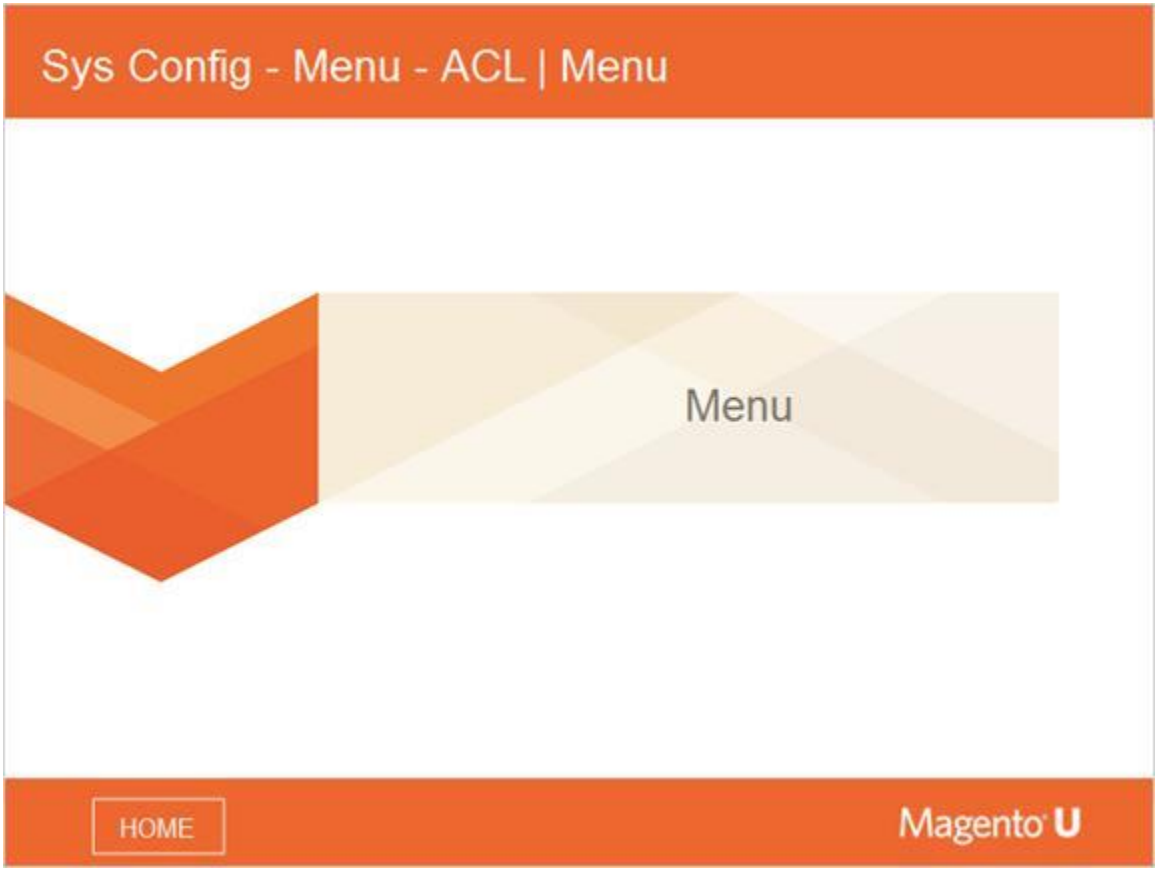
6.14 Reinforcement Exercise (6.6.1)



Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

6.15 Sys Config - Menu - ACL | Menu



Notes:

Next, we'll look at the Menu feature within Magento 2.0.

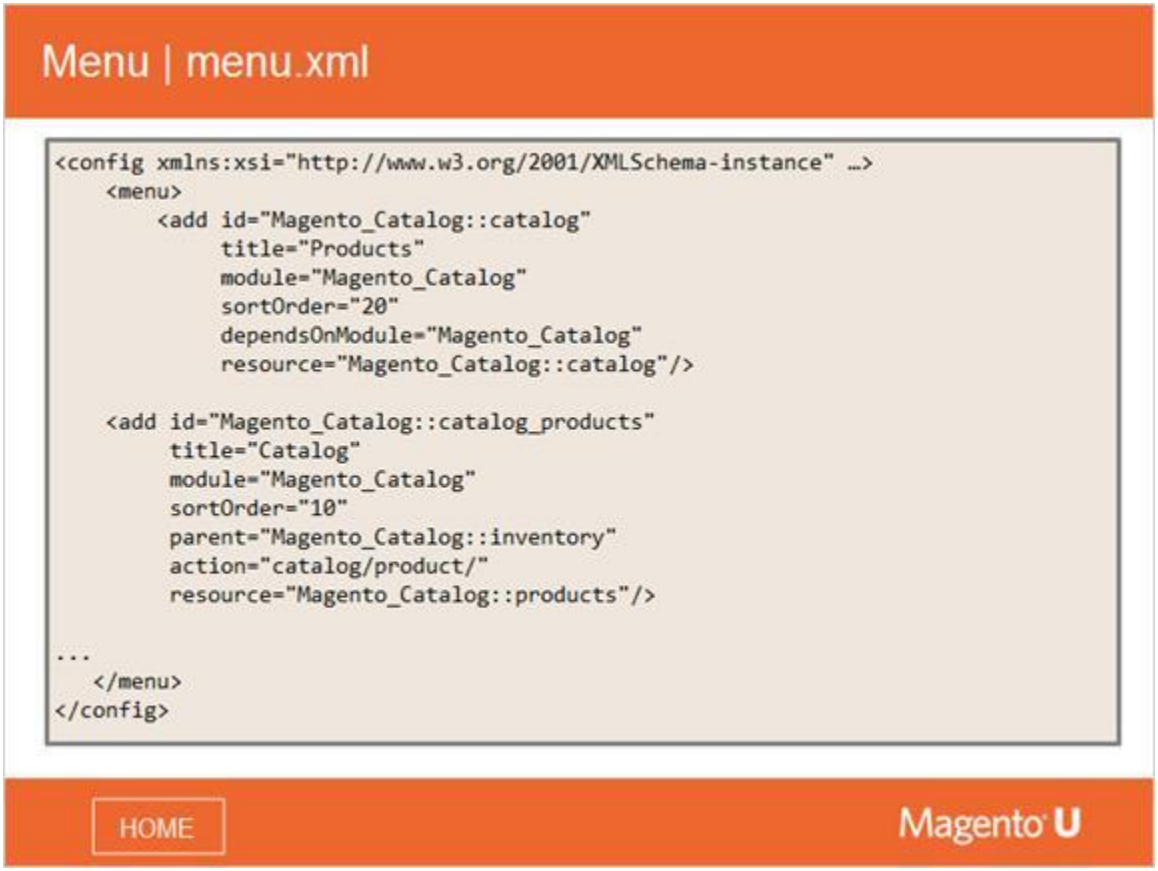
6.16 Menu | Overview

Notes:

Unlike Magento 1, the menu in Magento 2 is located on the left side of a page. This is done to simplify access to a menu from a tablet or smartphone.

Other than that, the menu structure in both Magento 1 and Magento 2 are the same.

6.17 Menu | menu.xml



Notes:

Menu is configured by the file `menu.xml`.

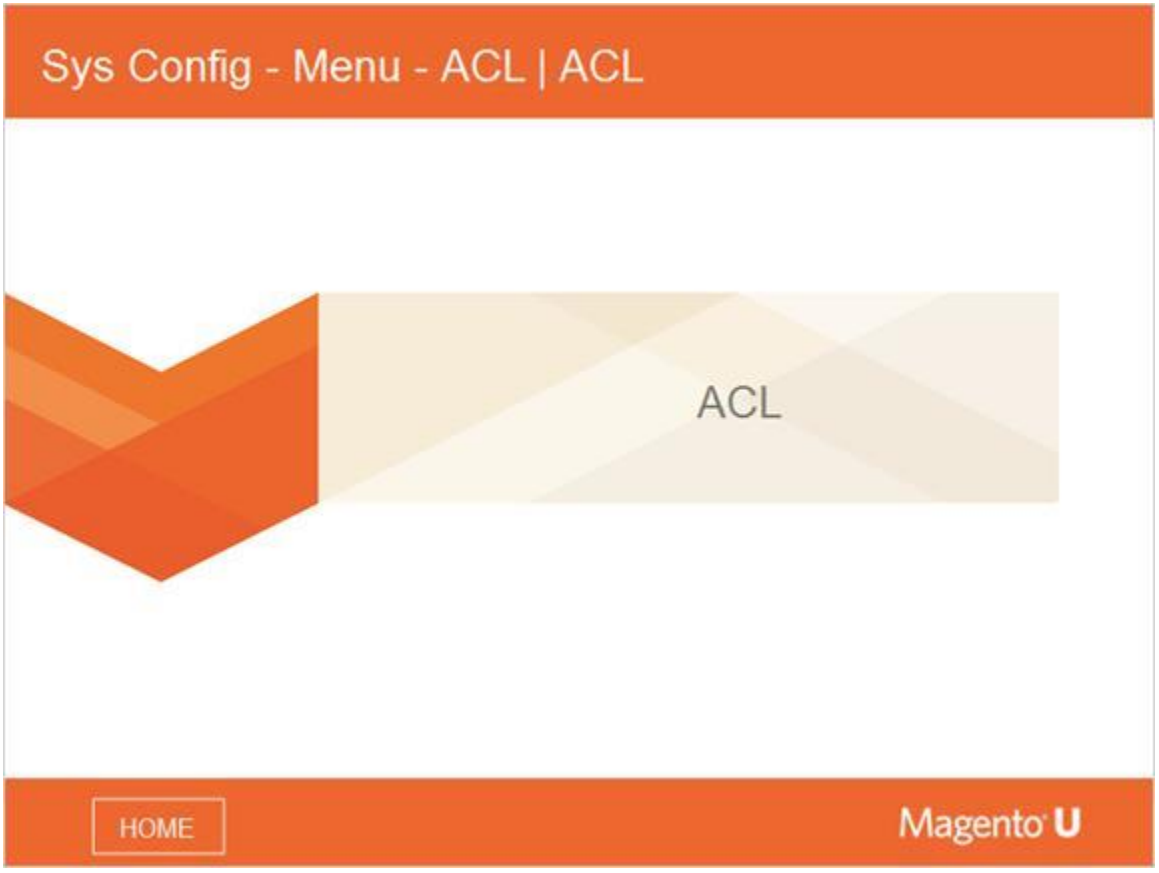
The code example shows how to typically add a new menu item, using an “add” directive. You should specify its name (`id` attribute), the `module` it belongs to, a visible `title`, and an ACL (`resource` attribute).

6.18 Reinforcement Exercise (6.6.2)

Notes:

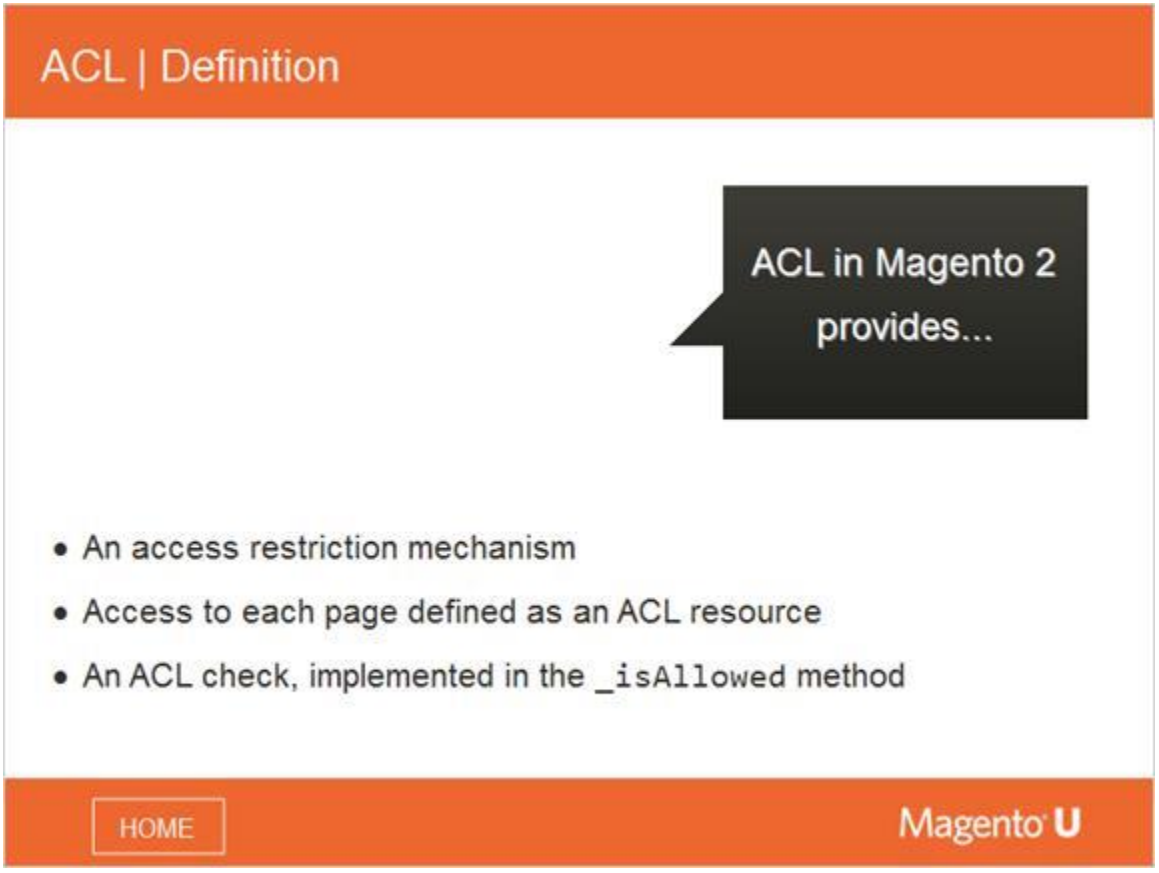
See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

6.19 Sys Config - Menu - ACL | ACL



Notes:

6.20 ACL | Definition

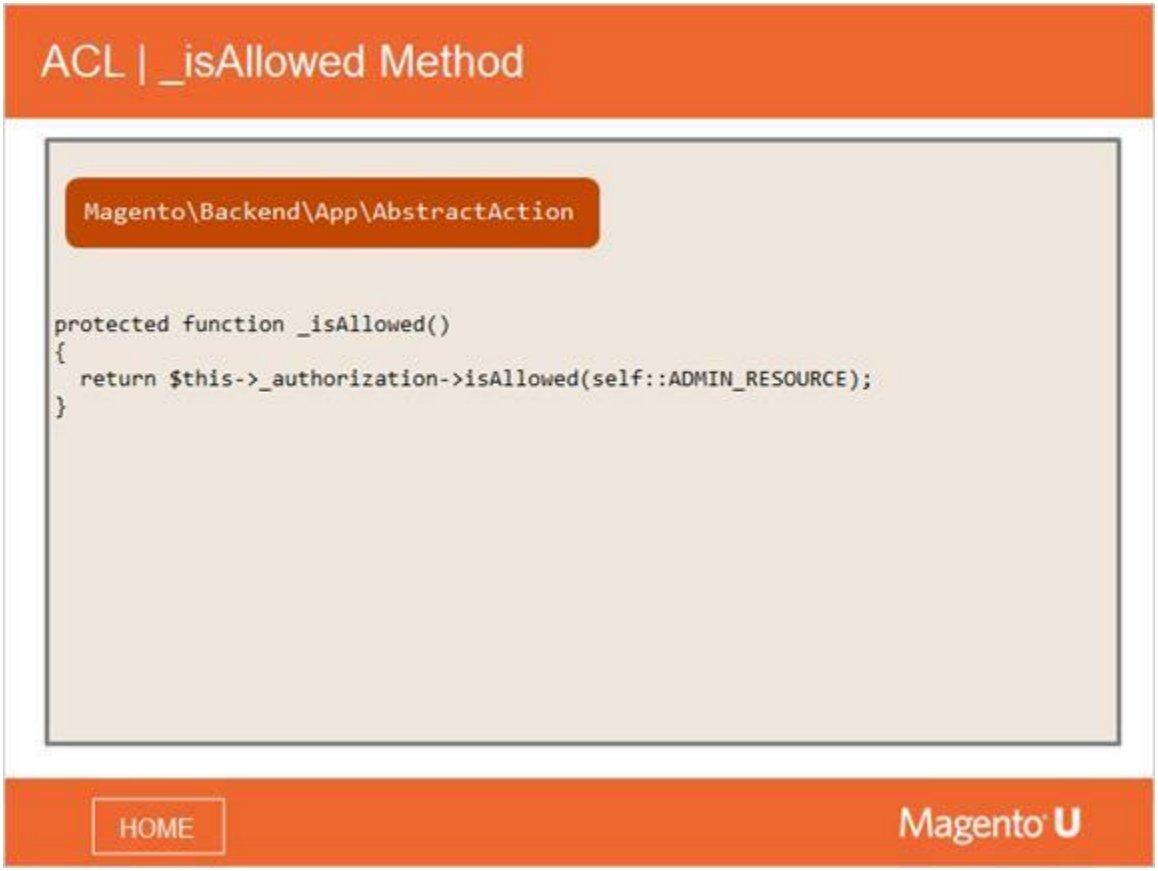


Notes:

The ACL system in Magento 2 is very similar to Magento 1.

It has a following structure:
- Each admin user is assigned a certain ACL role.
- Each role is assigned access to a specific set of resources.
- As in Magento 1, ACL verification is implemented on the controller level, using the `_isAllowed` method.

6.21 ACL | _isAllowed Method

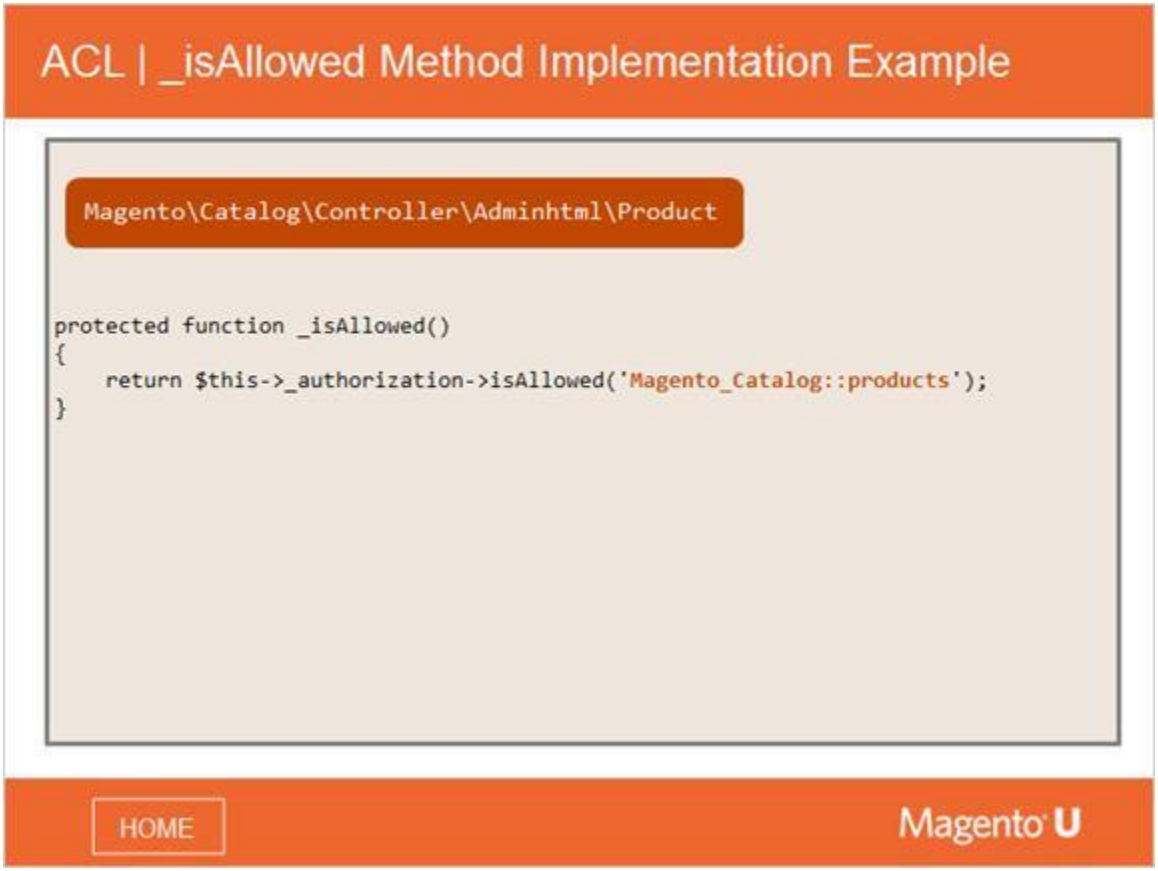


Notes:

The code presents an example of an `_isAllowed()` method.

You must implement this method for ACL verification to work.

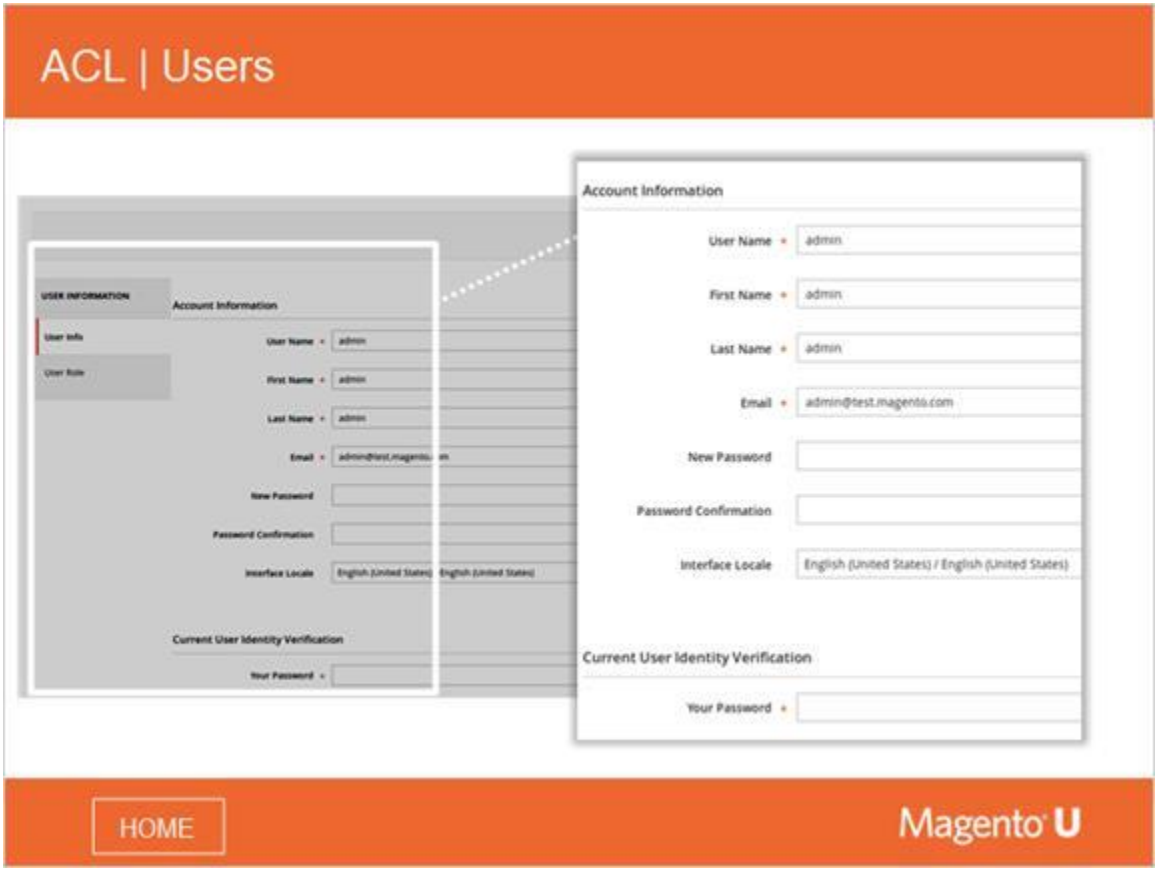
6.22 ACL | _isAllowed Method Implementation Example



Notes:

This code example is taken from the `Magento/Catalog` module.

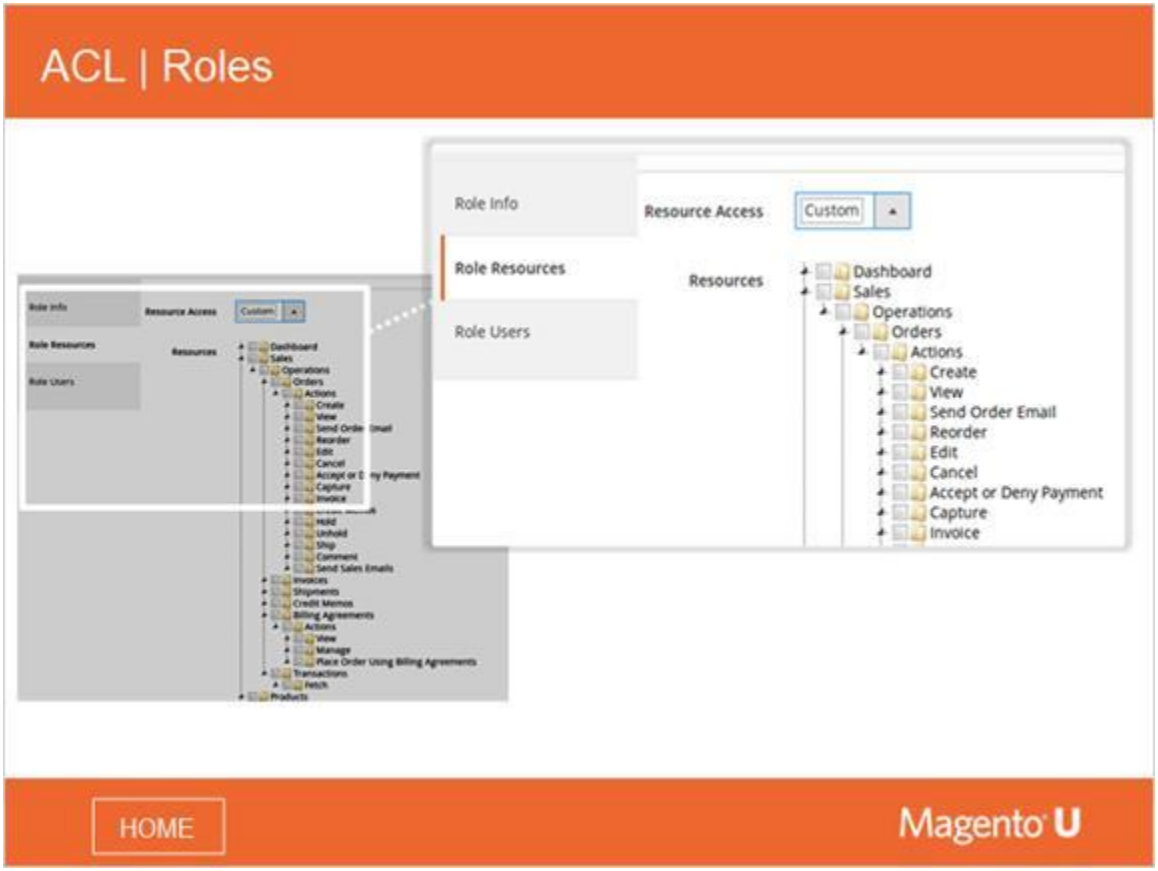
6.23 ACL | Users



Notes:

This image shows the admin user creation and management screen.

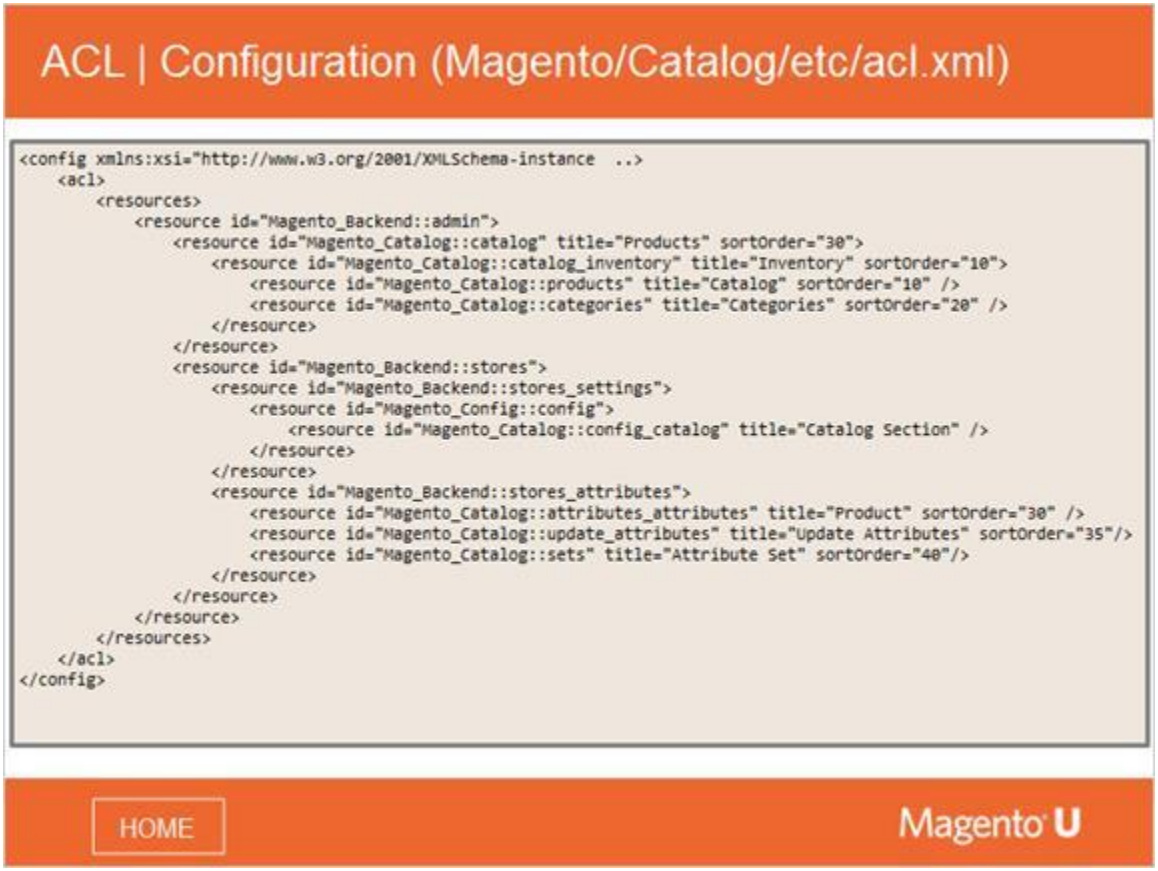
6.24 ACL | Roles



Notes:

Here is an ACL roles tree. The list of available resources is defined in an `acl.xml` configuration file.

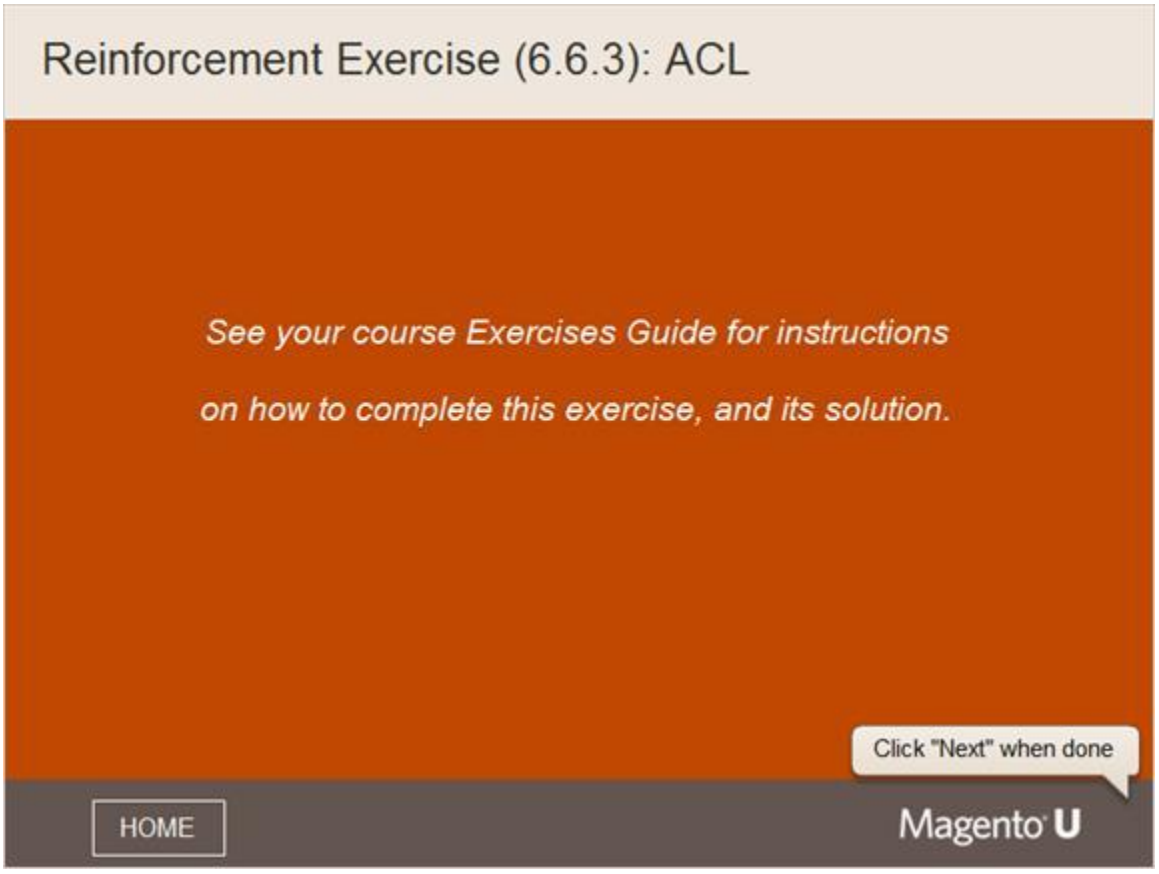
6.25 ACL | Configuration (Magento/Catalog/etc/acl.xml)



Notes:

This code provides an example of an `acl.xml` from the `Magento/Catalog` module.

6.26 Reinforcement Exercise (6.6.3)



Notes:

See your course Exercises Guide for instructions on how to complete this exercise, and the solution.

6.27 End of Unit Six